Sparse Approximate Inverse - A Massively Parallel Implementation

Emil Vilandt Rasmussen
nbz406
Implementation GitHub: https://github.com/nbz406/SPAI

June 12, 2023

Abstract

The Sparse Approximate Inverse (SPAI) algorithm calculates an approximate inverse to a matrix used to precondition a linear sytem to be solved by iterative solvers. The algorithm consists of independent sub-problems across the columns of the matrix and lends itself well to parallelization on GPU hardware. This project examines the theory of the algorithm, discusses how it can be implemented sequentially as well as in a GPU-parallel language. The project presents a sequential as well as a partially GPU parallel implementation that match the performance reported in the literature.

Contents

1 Introduction **3**
1.1 Computing an Effective Preconditioner 3
1.2 Contributions of this Project 5
1.2.1 Details of SPAI 5
1.2.2 Sequential and Parallel Implementations 5
1.2.3 Evaluation 6
1.3 Restrictions 6

2 Theory – The SPAI Algorithm 6
2.1 Initial Least Squares 6
2.2 Sparsity Pattern Updates 9
2.3 QR-updates in SPAI 12
2.4 The Complete Algorithm 15

3 Implementation 18
3.1 Initial Code Transformations 18
3.1.1 Loop Distribution 18
3.1.2 Loop Interchange 20
3.1.3 Coalesced Access and Irregular Problem Sizes 21
3.2 Parallel Implementation 22
3.2.1 Map-reduces 22
3.2.2 Caculating A_u and $r(I)$ 23
3.2.3 QR Decomposition using Householder Reflections . . . 23
3.2.4 Calculating $\tilde{J}$ 25
3.2.5 Post-SPAI Assembly of M 26
3.3 Edge Cases 27
3.4 Discussion 27
3.4.1 Improvements 27
3.4.2 Alternatives 28

4 Evaluation 28
4.1 Sequential Implementation 28
4.1.1 Correctness 29
4.1.2 Convergence 30
4.2 Parallel Implementation 32
4.2.1 Correctness 32

5 Further Work 32

6 Conclusion 33

A Appendix 36

1 Introduction

A parallel algorithm (SPAI) is presented for computing a sparse approximate inverse of a sparse $n \times n$ matrix, A. The algorithm consists solving n independent least squares problems for each column of A using QR factorization given some sparsity structure that captures the sparsity pattern of A^{-1}, such that the inverse can function as an effective preconditioner to solving a linear system $Ax = b$.

1.1 Computing an Effective Preconditioner

The problem of solving large, sparse linear systems of equations of the following form is widespread in fields such as engineering and machine learning.

$$Ax = b,\ x, b \in \mathbb{R}^n$$

Direct solvers have proven to be expensive in both the amount of work and storage required. Direct methods like Gaussian Elimination are also not very effective to implement in a parallel environment because of their sequential nature and are not effective for larger matrices due to their $O(n^3)$ running time. Iterative solvers such as GMRES, BCG, BI-CGSTAB and CG give an initial guess x_0 of the solution and iteratively compute better approximations x_i until a stopping criteria of $||b - Ax_i||/||b|| \leq tolerance$. Convergence is not guaranteed and may be very slow for these methods. To rectify this problem, a preconditioner to the system is introduced in the form of an approximate inverse to A, called M. The preconditioner can be computed as a left or right inverse with the (left) preconditioined system becoming:

$$AMy = b,\ \text{where } x = My,\ \text{or } MAx = Mb \tag{1}$$

Preconditioning the system has shown to make convergence of iterative solvers faster. In order for this timesave to be worth it and to reduce the overall computation time, the preconditioner must be computed in an efficient and parallel manner. The matrix-vector product My must also be calculated in each iteration of the iterative solver, so the number of nonzero entries in M should be relatively close to the number of nonzeros in A. Grote and Huckle presented an algorithm for computing the preconditioner in their 1995 paper: "Parallel Preconditioning with Sparse Approximate Inverses" [1]. Their algorithm (SPAI) minimizes the Frobenius norm of the error $AM - I$, which

leads to the inherently parallel algorithm of reducing the Frobenius norm of each column of the error, since the Frobenius norm of the error can be defined as:

$$||AM - I||_F^2 = \sum_{k=1}^{n} ||(AM - I)e_k||_2^2$$

The solution to the problem above can be split into n independent least squares problems that minimize the error of the n columns of the error:

$$\min_{m_k} ||Am_k - e_k||, k = 1, \ldots, n \tag{2}$$

where e_k is the k'th column of the identity matrix. These sub-problems could be solved explicitly, which would yield a small error, but also produce a sparsity structure that is very different from the actual inverse A^{-1}. A similar amount of non-zeros in M compared to A is required for the approximate inverse to be an effective preconditioner. The SPAI algorithm starts with an initial sparsity structure (often the identity matrix) and calculates m_k for that given sparsity structure using QR-decomposition and back substitution. Until a threshold of the norm of the residual is reached, the algorithm augments the sparsity structure and recalculates m_k using QR-updates. This algorithm is known as dynamic SPAI, where static SPAI have a fixed sparsity structure. The algorithm is parallel across columns and lends itself well to implementation on parallel hardware such as GPUs. Each step of the algorithm can be solved in parallel using efficient GPU-parallel batched dense matrix-operations.

There are three main criteria for the success of the SPAI algorithm. One, it should be fast to compute. This is required for the effort of computing a preconditioner to be worth it. This is measured by the time complexity of the algorithm and the amount of iterations it takes for each column to reach the stopping criteria. Two, it must produce a relatively good approximation of the actual inverse of A. This ensures faster convergence of iterative solvers on the preconditioned system. This criteria is measured by the Frobenius norm of the error: $||AM - I||_F^2$. Three, the sparsity structure of M must be close to A. This ensures that the matrix-vector product My from Equation 1 is not too expensive to compute, as it has to be computed in each iteration of the iterative solver. This criteria measured by the number of non-zeros of M relative to A. A ratio that is low or close to 1 is preferable. These criteria are what I will use to evaluate my implementation.

1.2 Contributions of this Project

There exist a number of papers on variations of the dynamic SPAI algorithm, but the available research on implementing it on GPU-hardware is very scarce. The goal of this thesis is to investigate the implementation details of the basic dynamic SPAI algorithm with QR-updates. I will present a sequential implementation and a partially parallel implementation as well as propositions for how to implement the remaining parts on parallel hardware, both of which I will evaluate the performance of.

1.2.1 Details of SPAI

The theory of the algorithm is well-described in the literature, but I have found implementation-specific details to be relatively lackluster. Usually only the mathematical representation of the algorithm is presented and crucial steps of the algorithm are left out. Most papers are not specific about what and how much actually needs to be computed, especially in regards to the QR-updates. My contribution is shedding more light on these details of the algorithm.

1.2.2 Sequential and Parallel Implementations

This project presents one approach to implementing the algorithm in a GPU-parallel language as well as presenting a simple sequential implementation. My parallel implementation produces the same preconditioner as my sequential implementation but performs magnitudes better despite not being fully parallel. Both produce error-norms and sparsity structures comparable with the results in [1]. Some steps of the parallel implementation remain sequential, but I will propose possible methods of implementation. These include the possibility of using the purely functional GPU- programming language with nested parallelism, Futhark, [2] [3]. The parallel implementation is done in the CUDA programming language [4], which is an extension to the C++ language that allows the user to write kernels that run on the GPU. I use the CUBLAS library [5] for dense matrix-operations and the CUB library [6] for sorting. The sequential implementation is done in Python [7] using functionality from Numpy [8] and Scipy [9].

1.2.3 Evaluation

My sequential implementation is evaluated on the three criteria: accuracy, sparsity of the result and speed of convergence. The results from my sequential implementation match the results reported by Grote and Huckle [1], indicating that the implementation is correct. The

1.3 Restrictions

Due to time restrictions, my implementation is only partially GPU-parallel, but most of the important matrix-operations are done in parallel using batched implementations provided by CUBLAS [5]. These cover matrix-matrix and matrix-vector multiplication as well as QR-decomposition using householder transformations. I propose solutions to parallelize the missing parts of the implementation. My partially parellel CUDA implementation has not worked for matrices with more than 5,000 columns due the memory required being very large. The CUDA implementation also has some issues that arise when A has empty columns. My stopping criteria for the CUDA version is also very naive and continues to update each column of M, until all columns fulfill the stopping criteria. This leads to excessive fill-in, and thus a high ratio of non-zeros in M compared to A.

2 Theory – The SPAI Algorithm

The SPAI algorithm consists of n independent sub-problems, that each calculate a column m_k of the preconditioning matrix M. I will present the algorithm for a single column and show how one could implement it sequentially in Python. My algorithm follows Grote and Huckle [1] closely with some slight modifications and some more explanations for the QR-updates.

2.1 Initial Least Squares

The algorithm starts with a given sparsity pattern of M, which in most cases is chosen to be the identity matrix I. A sparsity pattern for a given column refers to the row indices that contain non-zeros. For the identity this is k for column k of I. A least squares problem that minimizes the Frobenius norm is solved for this sparsity pattern as shown in Equation 2. Afterwards, the sparsity pattern is updated, by introducing new non-zeros in m_k, and the

least squares problem is solved again. This is done until a maximum fill-in is reached, or the norm of the error for the given column is low enough. In practice, the fill-in is kept low by limiting the amount of indices, that can be added to m_k in each iteration – usually 5 – and limiting the amount of iterations – usually 5 to 20.

To avoid calculating a minimization problem with n rows as in Equation 2, $\min_{m_k} ||Am_k - e_k||, k = 1, \ldots, n$, we can exploit that the matrix vector product Am_k only requires the multiplication of the columns of A with the non-zero values of m_k. We define J to be the set of row-indices of non-zero values in m_k. An example is shown below.

$$\underbrace{\begin{pmatrix} * & 0 & 0 & * & 0 \\ 0 & * & * & 0 & * \\ 0 & 0 & * & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & 0 \end{pmatrix}}_{A} \underbrace{\begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}}_{m_k}$$

The result of the multiplication also only has non-zeros in the row indices of the non-zero rows of $A(:, J)$:

$$\underbrace{\begin{pmatrix} 0 & 0 \\ * & * \\ 0 & * \\ * & 0 \\ 0 & 0 \end{pmatrix}}_{A(:,J)}$$

We define I to be the set of non-zero indices of $A(:, J)$. We define the sizes of the sets as $n_1 = |I|, n_2 = |J|$. This allows us to define a new reduced minimization problem to be solved for each column:

$$\min_{\hat{m}_k} ||\hat{A}\hat{m}_k - \hat{e}_k||, k = 1, \ldots, n \tag{3}$$

where $\hat{A} = A(I, J), \hat{m}_k = m_k(J), \hat{e}_k = e_k(I)$. This minimization problem is solved using QR-decomposition which decomposes $\hat{A}$ into a matrix-product of an upper triangular $n_2 \times n_2$ matrix R and an orthogonal matrix Q:

$$\hat{A}\hat{m}_k = \hat{e}_k \Longleftrightarrow QR\hat{m}_k = \hat{e}_k$$

Since Q is orthogonal, its inverse is equal to its transpose, and the solution to the minimization problem is found by solving the following upper triangular linear system:

$$R\hat{m}_k = Q^T \hat{e}_k(: n_2)$$

Since $\hat{e}_k$ is the k'th columm of the identity matrix in the row indices I, the operation $Q^T \hat{e}_k(: n_2)$ is simply a selection of the first n_2 elements of the k_I'th column of Q^T, where k_I is the index of k in I. Take $I = \{1, 2, 4\}$ and $k = 4$ as an example. I would produce an $\hat{e}_k$ that looks like this:

$$\underbrace{\begin{matrix} 1 \\ 2 \\ \\ k=4 \\ \\ \\ \\ \\ \end{matrix} \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}}_{e_k} \rightarrow \underbrace{\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}}_{\hat{e}_k}$$

which would then select the third column or Q^T. If I does not contain k, then $Q^T \hat{e}_k = \mathbf{0}_{n_1}$

Upper triangular systems can be solved in $O({n_2}^2)$ time using back substitution. It is worth noting that n_2 is relatively small and bounded by the amount of iterations of the SPAI algorithm and the number of indices added to J in each iteration. This means that n_2 at maximum is 1 + the number of iterations multiplied by the number of indices added per iteration, if you start with a sparsity pattern of the identity matrix. The QR-decomposition in my implementation is performed using Householder Reflections which is explained further in the implementation section.

Now that we have a column m_k that minimizes the error for the given sparsity structure, we can compute the residual and check whether or not the norm of the residual is below a set threshold, ε:

$$r = A(:, J)\hat{m}_k - e_k, \text{ stop if } ||r = A(:, J)\hat{m}_k - e_k|| < \varepsilon$$

However, since $A(:, J)\hat{m}_k$ only contains non-zeros in the row-indices in I, we do not need to compute the sparse matrix, dense vector product $A(:, J)\hat{m}_k$,

but we can get away with calculating the dense matrix, dense vector product which will be useful in the parallel implementation.

$$r(I) = \hat{A}\hat{m}_k - \hat{e}_k$$

$r(I)$ is then scattered to r using I. If k is not contained in I, then we need to subtract 1 from $r(k)$ in order to get the correct residual.

If $||r|| = ||A(:, J)\hat{m}_k - e_k|| < \varepsilon$, then we scatter $\hat{m}_k$ to $M[J, k]$. If we are not finished, the scatter operation is not performed. The Python implementation for the initial least squares could look like the code in Listing 1. You would also need a check for whether or not I includes k, but this is not included in the code snippet.

```
1    J = m_k.nonzero()[0] # gets row indices of non-zero elements
2    n2 = J.size
3    A_J = A[:,J]
4    I = np.unique(A_J.nonzero()[0])  # gets row indices of non-zero rows
5    n1 = I.size
6    Ahat = A[np.ix_(I, J)].todense()
7    Q, R = np.linalg.qr(Ahat)
8
9    QTe = np.matrix(Q.T[:,list(I).index(k)]).T
10   mhat_k = scipy.linalg.solve_triangular(R, QTe[0:n2])
11
12   e_k = np.matrix([0]*N).T
13   e_k[k] = 1
14   rI = Ahat * mhat_k - e_k[I] # dense matrix dense vector multiplication
15   r_norm = np.linalg.norm(rI) # calculate norm
16
17   r = np.zeros((M.shape[0],1)) # scatter
18   r[I] = rI
```

Listing 1: Python implementation for the initial least squares

2.2 Sparsity Pattern Updates

The rows where r is not equal to zero: $L = I \cup \{k\}$ are where we can improve upon the the approximation. We consider the non-zero column indices of $A(L, :)$ that are not already in J defined by J_{cand}. An example with $L = \{1, 5\}$ is shown below. The rows indices of L are highlighted with

red, and the non-zero columns of these rows are highlighted with blue.

$$\begin{array}{c} \\ 1 \\ \\ \\ \\ 5 \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & & & 4 & \end{array} \\ \begin{pmatrix} * & 0 & 0 & * & 0 \\ 0 & * & * & 0 & * \\ 0 & 0 & * & 0 & 0 \\ 0 & * & 0 & 0 & 0 \\ 0 & 0 & 0 & * & 0 \end{pmatrix} \end{array}$$

J_{cand} are candidates for being added to J to augment the sparsity structure of m_k. We want to calculate which indices of J_{cand} are the most profitable to add to J in terms of reducing the norm of the residual $||r||$. For each j in J_{cand}, we consider the following minimization problem:

$$\min_{\mu_j} ||r + \mu_j A(:, j)||$$

which is solved by

$$\mu_j = -\frac{r^T A(:, j)}{||A(:, j)||^2}$$

The norm ρ_j squared of the new residual $r + \mu_j A e_j$ is then

$${\rho_j}^2 = ||r||^2 + \mu_j A e_j = ||r||^2 - \frac{(r^T A(:, j))^2}{||A(:, j)||^2}$$

We select the most profitable indices – the indices with the lowest associated ρ_j^2 – usually a maximum of 5 to avoid unnecessary fill-in. Most indices produce the same ρ_j^2, so we only pick the ones that are smaller than the average. That is, we only pick an index j if

$${\rho_j}^2 \leq \frac{1}{|J_{cond}|} \sum_{i=1}^{|J_{cond}|} {\rho_i}^2$$

This heuristic also helps to reduce amount of unnecessary fill-in of m_k. The most profitable indices are contained in the set $\tilde{J}$. Using the augmented set $J^l = J^{l-1} \cup \tilde{J}^{l-1}$, where l indicates the iteration number, we could solve the minimization problem in Equation 3 with a new $I^l = I^{l-1} \cup \tilde{I}^{l-1}$, where $\tilde{I}$ is the nonzero rows of $A(: \tilde{J})$ not already contained in I. Grote and Huckle define $\tilde{I}$ to be the nonzero rows of $A(: J \cup \tilde{J})$ not already contained in I, but including J in the columns is redundant, since $I \cap \tilde{I} = \emptyset$ must hold. A Python implementation of the sparsity pattern updates could look like this:

```
while r_norm > epsilon and iter < maxiter:
        iter += 1
        L = np.union1d(I,k)
        Jcand = np.array([],dtype=int) #Calculate Jcand: All of the the new column indices of A that appear in all L rows but not in J
        for l in L:
            A_l = A[l,:]
            NZofA_l = np.unique(A_l.nonzero()[1])
            N_l = np.setdiff1d(NZofA_l, J)
            Jcand = np.union1d(Jtilde,N_l)

        avg_rho = 0
        j_rho_pairs = []
        for j in Jcand:
            Ae_j = A[:,j].todense()
            Ae_jnorm = np.linalg.norm(Ae_j)
            rTAe_j = r.T * Ae_j
            rho_jsquared = r_norm*r_norm - (rTAe_j * rTAe_j) / (Ae_jnorm * Ae_jnorm)
            avg_rho += rho_jsquared
            j_rho_pairs.append((rho_jsquared[0,0],j))
        avg_rho = avg_rho / len(j_rho_pairs)

        heap = [] # Creates min heap to quickly find indices with lowest error.
        for pair in j_rho_pairs:
                heapq.heappush(heap, (pair[0], pair[1]))

        pops = 0
        Jtilde = [] # Select the 5 indices with rho below average that create the lowest residuals
        while len(heap) > 0 and pops < n_most_profitable_indices:
            pair = heapq.heappop(heap)
            if (pair[0] < avg_rho):
                Jtilde.append(pair[1])
                pops += 1

        Jtilde = np.sort(Jtilde)
        n2tilde = len(Jtilde)

        Itilde = np.setdiff1d(np.unique(A[:,Jtilde].nonzero()[0]), I)
        n1tilde = len(Itilde)
```

Listing 2: Python implementation for the sparsity pattern updates

We could then solve the minimization problem for the augmented sparsity structure as in the previous section and continue for a set number of iterations, or until the $||r|| < \varepsilon$. However, we can save great amount of work by utilizing the fact that we have already computed the QR-decomposition for a subset of the columns, J, and rows, I, of this new problem.

2.3 QR-updates in SPAI

The algorithm presented by Grote and Huckle uses QR-updates, but their description fails to mention some significant details. Specifically the fact that you are solving a system where the rows and columns are permuted, which is noted by Matous Sedlacek in [10]. However, this, and other papers, are not specific about what is needed to compute the updated Q and R. We wish to solve the updated linear system of equations:

$$\min_{m_k(J\cup\tilde{J})} \| A(I\cup\tilde{I}, J\cup\tilde{J})m_k(J\cup\tilde{J}) - e_k(I\cup\tilde{I}) \|, k = 1,\dots,n$$

There exist permutation matrices, P_r, P_c, such that:

$$P_r A(I\cup\tilde{I}, J\cup\tilde{J})P_c = \tilde{A} = \begin{pmatrix} A(I,J) & A(I,\tilde{J}) \\ A(\tilde{I},J) & A(\tilde{I},\tilde{J}) \end{pmatrix}$$

The permutation matrices permute the new indices to the right and down in the matrix. This means that we can solve an modified system of equations:

$$\min_{\underbrace{{P_c}^T m_k(J\cup\tilde{J})}_{\tilde{m}_k}} \| \underbrace{P_r A(I\cup\tilde{I}, J\cup\tilde{J})P_c}_{\tilde{A}} \underbrace{{P_c}^T m_k(J\cup\tilde{J})}_{\tilde{m}_k} - \underbrace{P_r e_k(I\cup\tilde{I})}_{\tilde{e}_k} \|, k = 1,\dots,n$$

and extract the original solution $m_k(J\cup\tilde{J}) = P_c{P_c}^T m_k(J\cup\tilde{J}) = P_c\tilde{m}_k$. We do not have to calculate P_c and P_r, if we realize that the permutations are given implicitly by treating the union operation $\cup$ as an concatenation operation $\frown$ that preserves the order of J and $\tilde{J}$ in $J\frown\tilde{J}$. Thus we can write:

$$\begin{aligned} \tilde{A} &= P_r A(I\cup\tilde{I}, J\cup\tilde{J})P_c = A(I\frown\tilde{I}, J\frown\tilde{J}) = \begin{pmatrix} A(I,J) & A(I,\tilde{J}) \\ A(\tilde{I},J) & A(\tilde{I},\tilde{J}) \end{pmatrix} \\ \tilde{m}_k &= {P_c}^T m_k(J\cup\tilde{J}) = m_k(J\frown\tilde{J}) \\ \tilde{e_k} &= P_r e_k(I\cup\tilde{I}) = e_k(I\frown\tilde{I}) \end{aligned}$$

The end result, we are interested in is scattering the result $m_k(J\cup\tilde{J})$ to m_k using $J\cup\tilde{J}$, but we can solve for $m_k(J\frown\tilde{J})$ and simply scatter the result to m_k using $J\frown\tilde{J}$.

The form of $\tilde{A}$ allows us to perform a QR-update instead of calculating the whole decomposition of $\tilde{A}$, [11], [12]. We first realize that $A(\tilde{I},J) = \mathbf{0}$,

since all of the non-zero rows if $A(:,J)$ are included in I, and $I \cap \tilde{I} = \emptyset$. Multiplying $\tilde{A}$ by the identity:

$$I_{n_1+\tilde{n}_1} = \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} Q^T & \\ & I_{\tilde{n}_1} \end{pmatrix}$$

where $\tilde{n}_1 = |\tilde{I}|$, and $\tilde{n}_2 = |\tilde{J}|$ gives us:

$$\begin{aligned} \tilde{A} &= \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} Q^T & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} A(I,J) & A(I,\tilde{J}) \\ 0 & A(\tilde{I},\tilde{J}) \end{pmatrix} \\ &= \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} Q^T A(I,J) & Q^T A(I,\tilde{J}) \\ 0 & A(\tilde{I},\tilde{J}) \end{pmatrix} \end{aligned}$$

By the definition of $A(I,J) = QR$, we get $Q^T A(I,J) = R$:

$$\tilde{A} = \begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} R & Q^T A(I,\tilde{J}) \\ 0 & A(\tilde{I},\tilde{J}) \end{pmatrix} \iff \tag{4}$$

$$\begin{pmatrix} Q^T & \\ & I_{\tilde{n}_1} \end{pmatrix} \tilde{A} = \left(\begin{array}{c|c} R & A_u\,[:n_2,:] \\ \hline 0 & A_u\,[n_2:,:] \\ 0 & A(\tilde{I},\tilde{J}) \end{array}\right) \iff \tag{5}$$

$$\begin{pmatrix} Q^T & \\ & I_{\tilde{n}_1} \end{pmatrix} \tilde{A} = \left(\begin{array}{c|c} R & B_1 \\ \hline 0 & B_2 \end{array}\right) \tag{6}$$

Where $A_u = Q^T A(I,\tilde{J})$. The first n_2 columns of the right-hand side matrix are already upper triangular. Applying householder transformations to make the remaining $n_1 + \tilde{n}_1 - n_2$ columns upper triangular will involve multiplying both sides with

$$\begin{pmatrix} I_{n_2} & \\ & Q_B^T \end{pmatrix}$$

where Q_B is the orthogonal matrix that transforms the bottom right square, denoted $B_2 = Q_B R_B$, to upper triangular form in Equation 6. The upper right square, denoted B_1 remains unchanged. The result is our new updated QR-decomposition of $\tilde{A}$:

$$\begin{pmatrix} I_{n_2} & \\ & Q_B^T \end{pmatrix} \begin{pmatrix} Q^T & \\ & I_{\tilde{n}_1} \end{pmatrix} \tilde{A} = \left(\begin{array}{c|c} R & B_1 \\ \hline 0 & R_B \end{array}\right) \iff$$

$$\tilde{A} = \underbrace{\begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{n_2} & \\ & Q_B \end{pmatrix}}_{Q_{new}} \underbrace{\left(\begin{array}{c|c} R & B_1 \\ \hline 0 & R_B \end{array}\right)}_{R_{new}}$$

The resulting QR-decomposition $\tilde{A} = Q_{new}R_{new}$ is used to solve $\min_{\tilde{m}_k} \parallel \tilde{A}\tilde{m}_k - \tilde{e}_k \parallel$ with back substitution, since R_{new} is upper triangular.

$$\tilde{m}_k = R_{new}^{-1}(Q_{new}^T \tilde{e}_k)[: n_2 + \tilde{n}_2]$$

The matrix-matrix product that defines Q_{new} does not need to be calculated explicitly, since the result is:

$$\begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{n_2} & \\ & Q_B \end{pmatrix} = \begin{pmatrix} Q[:,: n_2] & Q[:, n_2 :]Q_B[: n_2, :] \\ 0 & Q_B[n_2 :, :] \end{pmatrix}$$

and you just apply the householder transformations used to transform B_2 to the last $n_1 + \tilde{n}_1 - n_2$ columns of $\begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix}$. If the norm residual is still greater than ε, you update the index sets with l denoting the iteration number $I_l = I_{l-1} \frown \tilde{I}_{l-1}$ and $J_l = J_{l-1} \frown \tilde{J}_{l-1}$ and $n_{1_l} = n_{1_{l-1}} + \tilde{n}_{1_{l-1}}$, $n_{2_l} = n_{2_{l-1}} + \tilde{n}_{2_{l-1}}$, update $Q_l = Q_{new}$, $R_l = R_{new}$ and perform the sparsity pattern update and QR-update again. In practice, R_l is stored in place in $\tilde{A}_l$ along with the householder vectors that construct Q_l and transform $\tilde{A}_l$ into R_l. To get a better understanding of how $\tilde{A}$ and thus R_l is updated in each iteration, I have provided the following diagram:

$$\tilde{A}^l = \left(\begin{array}{c|c} \begin{array}{c|c} \tilde{A}^{l-2} & A(I^{l-1}, \tilde{J}^{l-1}) \\ \hline 0 & A(\tilde{I}^{l-1}, \tilde{J}^{l-1}) \end{array} & A(I^l, \tilde{J}^l) \\ \hline 0 & A(\tilde{I}^l, \tilde{J}^l) \end{array}\right)$$

$$R^l = \left(\begin{array}{c|c} \begin{array}{c|c} R^{l-2} & B_1^{l-1} \\ \hline 0 & R_B^{l-1} \end{array} & B_1^l \\ \hline 0 & R_B^l \end{array}\right)$$

The QR-update can be performed sequentially Python as shown in Listing 3:

```
1    AIJtilde = A[np.ix_(I, Jtilde)]
```

```
    AItildeJtilde = A[np.ix_(Itilde,Jtilde)]

    QTAIJtilde = Q.T * AIJtilde
    B_1 = QTAIJtilde[:n2,:]
    B_2 = np.vstack((QTAIJtilde[n2:n1,:], AItildeJtilde.todense()))

    QB, RB = np.linalg.qr(B_2, mode="complete")
    RB = RB[:n2tilde,:n2tilde]

    R = np.hstack((np.vstack((R, np.zeros((n2tilde, n2)))), np.vstack((B_1,
    RB))))
    q = np.hstack((np.vstack((Q[:,n2:], np.zeros((n1tilde,n1-n2)))), np.
    vstack((np.zeros((n1,n1tilde)), np.identity(n1tilde)))))
    Q = np.hstack((np.vstack((Q[:,:n2], np.zeros((n1tilde,n2)))), q * QB))

    J = np.append(J,Jtilde)
    I = np.append(I,Itilde) # Itilde
    n2 = J.size
    n1 = I.size
```

Listing 3: Python implementation for the QR-update

2.4 The Complete Algorithm

The complete SPAI algorithm can be summarized as follows, given an initial sparsity structure M. The initial least squares is shown in Algorithm 1 and the iterative part of SPAI is shown in Algorithm 2.

Algorithm 1 Initial SPAI for a single column k

$m_k \leftarrow M[:, k]$
$J \leftarrow$ indices non-zero elements of m_k, $n_2 \leftarrow |J|$
$I \leftarrow$ row indices of non-empty rows of $A(:, J)$, $n_1 \leftarrow |I|$
$Q, R \leftarrow$ QR-decompose$(A(I, J))$
if $k \in I$ **then**
  $\hat{c}_k \leftarrow$ first n_2 elements of the k_I'th column of Q^T, where k_I is the index of k in I
else
  $\hat{c}_k \leftarrow \mathbf{0}$
end if
$\hat{m}_k \leftarrow$ Solve upper triangular system $R\hat{m}_k = \hat{c}_k$
$r(I) \leftarrow A(I, J)\hat{m}_k - e_k(I)$
if $k \notin I$ **then**
  $r(k) -= 1$
end if

Algorithm 2 SPAI sparsity pattern updates and QR-updates for column k

$iter \leftarrow 0$
while $iter < maxiter$ and $\| r \| > \varepsilon$ **do**
  iter++
  $J_{cand} \leftarrow$ indices of non-empty columns in $A(\{k\} \cup I, :)$
  $\tilde{J} \leftarrow$ maximum of 5 indices from J_{cand} that produce lowest ${\rho_j}^2 = ||r||^2 - \frac{(r^T A(:,j))^2}{||A(:,j)||^2}$ that are also below the average ${\rho_j}^2$, $\tilde{n}_2 \leftarrow |\tilde{J}|$
  $\tilde{I} \leftarrow$ row indices of non-empty rows of $A(:, J)$ not already in I, $\tilde{n}_1 \leftarrow |\tilde{I}|$
  $A_u \leftarrow Q^T A(I, \tilde{J})$
  $B_1 \leftarrow A_u(: n_2, :)$
  $B_2 \leftarrow \begin{pmatrix} A_u(n_2 :, :) \\ A(\tilde{I}, \tilde{J}) \end{pmatrix}$
  $Q_B, R_B \leftarrow$ QR-decompose(B_2)
  $R \leftarrow \left(\begin{array}{c|c} R & B_1 \\ \hline 0 & R_B \end{array}\right)$
  $Q \leftarrow \begin{pmatrix} Q[:, : n_2] & Q[:, n_2 :]Q_B[: n_2, :] \\ 0 & Q_B[n_2 :, :] \end{pmatrix}$
  $J \leftarrow J \frown \tilde{J}$, $n_2 \leftarrow n_2 + \tilde{n}_2$
  $I \leftarrow I \frown \tilde{I}$, $n_1 \leftarrow n_1 + \tilde{n}_1$
  if $k \in I$ **then**
    $\tilde{c}_k \leftarrow$ first n_2 elements of the k_I'th column of Q^T, where k_I is the index of k in I
  else
    $\tilde{c}_k \leftarrow \mathbf{0}$
  end if
  $\tilde{m}_k \leftarrow$ Solve upper triangular system $R\tilde{m}_k = \tilde{c}_k$
  $r(I) \leftarrow A(I, J)\hat{m}_k - e_k(I)$
  if $k \notin I$ **then**
    $r(k) -= 1$
  end if
end while
$M(J, k) \leftarrow \tilde{m}_k$

3 Implementation

This section contains a description of how one could implement a GPU-parallel SPAI algorithm along with arguments for the correctness of the transformations required. I will start with a general description and move on to more specific kernels later. Both my sequential Python implementation and the partially parallel CUDA implementation can be found at https://github.com/nbz406/SPAI. The CUDA code is self-contained within `SPAI/CudaRuntimeTest/CudaRuntimeTest/kernel.cu`, and the Python implementation can be found at `SPAI/SpaiTest.py`. The structure of the code is not very clean due to time restrictions.

3.1 Initial Code Transformations

Before one can write GPU-parallel kernels, some transformations need to be applied to the code that preserve the semantics of the program but make its structure more suited for GPU execution. The matrices A and M are stored in a compressed sparse column-major (CSC) format, and A is also kept in compressed sparse row-major (CSR) format for easy access to rows. All the dense matrices are stored in a flat column-major format.

3.1.1 Loop Distribution

To begin the parallel implementation, we must understand the structure of the algorithm. This will allow us to apply safe transformations to the sequential implementation, that preserve the semantics of the algorithm while allowing for easier parallelization. Assuming that we have a sequential implementation in CUDA, the general structure of the program would look something like this:

```
for (int k = 0; k < n_cols; k++) { // parallel loop
    // perform initial least squares
    J = nonzero_indices(M[:,k]);
    Statement 1;
    Statement 2;
    ...

    while (iterations < max_iter && norm(r) > epsilon) {
        // Perform sparsity pattern updates
        // and solve linear system using QR-updates
        Statement 3;
        ...
    }
```

```
14
15      // Distribute m_k to sparse representation of M(J,k)
16      Statement 4;
17      ...
18      free(allocations);
19  }
```

Listing 4: Initial sequential CUDA implementation

The algorithm consists of n independent sub-problems, which means that every iteration of the outer for-loop is independent of one another; it is a parallel loop. Using the fact that a parallel loop may be distributed across each of its inner statements [13], we can transform the code to the following:

```
// perform initial least squares
for (int k = 0; k < n_cols; k++) { // parallel loop
    J = nonzero_indices(M[:,k]);
}
for (int k = 0; k < n_cols; k++) { // parallel loop
    Statement 1;
}
for (int k = 0; k < n_cols; k++) { // parallel loop
    Statement 2;
}
...

for (int k = 0; k < n_cols; k++) { // parallel loop
    while (iterations < max_iter && norm(r) > epsilon) {
        // Perform sparsity pattern updates
        // and solve linear system using QR-updates
        Statement 3;
        ...
    }
}
// Distribute m_k to sparse representation of M(J,k)
for (int k = 0; k < n_cols; k++) { // parallel loop
    Statement 4;
}
...
```

Listing 5: Outer loop distributed

Prior to the distribution of the loop over the individual statements, we could keep a single array for each of our data-structures, J, for example. However, distributing the loop requires us to keep each J_k stored in a way, that is accessible between loops. This is done by array expansion, which takes all scalars and arrays used between statements and expands them with an array dimension equal to the size of the outer loop. Each access is then replaced with an index into this large array based on the size of each sub-array [13].

The while loop will run for a different number of iterations for each subproblem. This poses an issue, since we cannot further distribute the outer parallel loop inside the while loop. Due of a lack of time, I have chosen the simple strategy of letting the loop run until the max number of iterations has been reached, or until all columns have converged, that is, if $\bigwedge_{k=0}^{n} ||r_k|| < \varepsilon$. As we will see in the evaluation section, this is a poor strategy, since a good amount of columns do not converge, which makes the while loop run for the full number of iterations causing unnecessary fill-in.

A simple alternative to this approach is to resize the parallel loop in each iteration of the while loop and only computing the columns that have not converged. This would require scattering the arrays and matrices of the unfinished columns to a smaller array while keeping track of the column number. This should speed up each kernel. However, if the amount of unfinished columns is very small, we could run into the problem of there not being enough parallelism to fully utilize the GPU.

3.1.2 Loop Interchange

With either of the choices made above, we can exploit the fact that the while loop will always run for the same number of iterations; either we perform the body of the while loop for all columns as in my implementation, or we resize the parallel loop to be `n_cols = n_cols - n_cols_finished`. Since the while loop is now essentially a sequential for loop of fixed size nested perfectly inside the outer parallel for loop, we can perform loop interchange. From Corollary 2 of [13], in a perfect loop nest, it is always safe to interchange a parallel loop inwards one step at a time. The outer loop is interchanged inwards and further distributed across its inner statements becoming:

```
while (iterations < max_iter) {
    for (int k = 0; k < n_cols; k++) { // parallel loop
        Statement 1;
    }
    for (int k = 0; k < n_cols; k++) { // parallel loop
        Statement 2;
    }
    ...
    // Only perform work on non-converged columns
    n_cols = n_cols - n_cols_finished
    // scatter remaining data structures into smaller sized array
}
```

Loop interchange can also be used to make data accesses more efficient by letting the index that accesses the innermost dimension of an array come

from the inner loop.

3.1.3 Coalesced Access and Irregular Problem Sizes

One of the most important parts of writing efficient GPU-parallel code is coalesced access, which refers to a pattern where each thread of the GPU executing in lockstep accesses consecutive memory addresses in global memory in a load or store operation. Sequential code is often not written with this in mind, so we may need to perform some transformations. Take the following pseudocode for matrix-vector multiplication of a row-major matrix `A` and vector `v`, which has good spatial locality if run on a CPU.

```
for (int i = 0; i < m; i++) { // parallel
    res[i] = 0;
    for (int j = 1; j < n; j++) { // sequential
        res[i] += A[i * n + j] * v[j]; // row major access
    }
}
```

Assigning a thread for each of the m rows, each of which executes the inner loop, the SIMD load instruction that reads `A[i * n + j]` will read elements that are `n` elements apart in memory. This is uncoalesced access. The problem can be rectified by instead transposing `A` and accessing it in column-major format. Now each thread accesses consecutive elements, making reads and writes very efficient on the GPU.

```
for (int i = 0; i < m; i++) { // parallel
    res[i] = 0;
    for (int j = 1; j < n; j++) { // sequential
        res[i] += A[j * m + i] * v[j]; // column major access
    }
}
```

Transposition requires that each sub-array has the same dimensionality. For each column of the SPAI algorithm, there is no guarantee that the problem sizes are the same. This means that, if we want to transpose and get coalesced access, we must pad our sub-arrays with data to make all the dimensions match the dimensions of the largest possible sub-array. Therefore, I allocate enough space for n of the largest possible arrays and matrices that will exist across the columns. Each A matrix will have size $\max_k\{n_{1_k}\} \cdot \max_k\{n_{2_k}\}$, etc. For matrices A and Q, I pad the extra memory with zeros and with ones on the diagonal. Padding with ones on the diagonal ensures that the back-substitution for solving the linear system does not contain divisions by

zero, and when Q is constructed, it is initialized to be the identity. My subarrays for I and J are padded with some large value, specifically n, since I am sorting them and they can only contain values up to $n-1$ with zero-based indexing.

3.2 Parallel Implementation

Now that the sequential CUDA-code consists of a number of for loops over the n columns, we can look at some specific ways to parallelize parts of the algorithm.

A big concern when writing GPU-parallel code is memory reuse. Allocations and frees on the GPU are expensive and we want to keep them to a minimum. Since we already know that the size of J_k at maximum will be the maximal sparsity pattern that we allow, we can allocate a single array for the n arrays of J_k at once and reuse the memory in each iteration. The size of this array is usually $\max\{nnz(m_k)\} = nnz(m_k)_{iteration0} + maxiter \times \max\{|\tilde{J}|\} = 1 + 10 \times 5$. During the implementation, I worked under the assumption, that we could not determine a maximum size of I_k beforehand, and as a result the sizes of the $\tilde{A}$ and Q. Therefore, I chose to limit the amount of allocations by doubling the size of a given array until it had enough space for the new problem size. This ensured that I would at maximum perform $O(\log_2 \text{max array size needed})$ allocations. In practice this was very few. However, we can propose an upper bound on $|I_k|$ as I mention in the discussion, which would completely eliminate the extra allocations.

For all the major matrix-operations, I use the batched functions from the CUBLAS library[5]. This includes matrix-matrix multiplication, matrix-vector multiplication, QR-decomposition and solving upper triangular systems. I also use the segmented sort functionality from [6].

3.2.1 Map-reduces

Some of the the loops that I have not yet parallelized can be written in terms of a nested map-reduce operations. An example is the calculation of the norm of each column of the original matrix A shown below:

```
1 for (int k = 0; k < n_cols; k++) // map (^2) reduce (+)
2     {
3         A_col_norms[k] = 0;
4         const int c_ptr = csc_col_ptr_A[k];
5   const int c_size = csc_col_ptr_A[k + 1] - c_ptr;
```

```
6         for (int i = 0; i < c_size; i++)
7         {
8             const double v = csc_val_A[c_ptr + i];
9             A_col_norms[k] += v*v;
10        }
11        A_col_norms[k] = sqrt(A_col_norms[k]);
12    }
```

The inner map-reduce would be mapped on each column by an outer map. The inner map would map the squaring function to each row element followed by a reduction using the addition operator. Lastly a map of the square root function on each column would produce the desired result. This sort of nested parallel code can be written in a functional style in Futhark [2], [3], which can be compiled to CUDA code and linked to the larger parallel application.

3.2.2 Caculating A_u and $r(I)$

$A_u = Q^T A(I, \tilde{J})$ and $r(I) = \tilde{A}\tilde{m}_k - \tilde{e}_k$ can both be performed using CUBLAS batched matrix-matrix and matrix-vector multiplication. For $r(I) = \tilde{A}\tilde{m}_k - \tilde{e}_k$, I first initialize $r(I) = \tilde{e}_k$, and then perform $r(I) = \tilde{A}\tilde{m}_k - r(I)$, since the CUBLAS matrix-vector multiplication has the capability of performing an update on the input vector. This eliminates the need for a separate negation kernel.

3.2.3 QR Decomposition using Householder Reflections

During SPAI, for each column we have to perform a QR-decomposition of $\underset{n_1 \times n_2}{A}$ in the initial least squares as well as a QR-decomposition of $\underset{n_1+\tilde{n}_1-n_2\times\tilde{n}_2}{B_2}$ each iteration of the while loop. The QR-decomposition can be performed using Householder reflections. I use CUBLAS' batched QR-decomposition, which transforms n matrices of the same size A to upper triangular form R in place using householder transformations. It stores the householder vectors used for the transformation in the lower-triangular part of A. Since the SPAI algorithm uses QR-updates, we need to calculate Q explicitly such that it can be used for the QR-update in the next iteration. To do this, I use part of the Householder QR algorithm presented in [14], which, through a series of householder transformations, converts a given $m \times n$ matrix A to upper triangular form, which is the R of the QR decomposition:

$$R = P_{n-1} \dots P_2 P_1 A$$

Here $P_i = I - \frac{2}{{v_i}^T v_i} v_i {v_i}^T$, and v_i is a householder vector. The algorithm also computes the orthogonal matrix $Q = P_1^T P_2^T \dots P_{n-1}^T$, which is what we are interested in. The algorithm for constructing Q be summarized as follows in pseudocode assuming that we already have the householder vectors v_i and $\beta = \frac{2}{{v_i}^T v_i}$, stored in the lower triangular part of `A` and in `betas` respectively.

```
Q = Identity(m);
for (int k = 0; k < n; k++) { // sequential
    v = A[k:m,k];
    Q[:m, k:m] = Q[:m, k:m] - betas[k] * Q[:m, k:m] * v * v^T;
}
```

The contents of the loop can be broken down into matrix-vector multiplication of Q and v, a scalar-vector multiplication of β and Qv and a rank-1 update of Q:

```
Q = Identity(m);
for (int k = 0; k < n; k++) { // sequential
    v = A[k:m,k];
    Qv =  Q[:m, k:m] * v;
    betaQv = betas[k] * Qv;
    Q[:m, k:m] = Q[:m, k:m] - betaQv * v^T;
}
```

Since we have to perform this for each of our `n_cols` columns, we get:

```
for (int kk = 0; kk < n_cols; kk++) { // parallel
    Q = Identity(m);
    for (int k = 0; k < n; k++) { // sequential
        v = A[k:m,k];
        Qv =  Q[:m, k:m] * v;
        Qv = betas[k] * Qv;
        Q[:m, k:m] = Q[:m, k:m] - betaQv * v^T;
    }
}
```

Performing array expansion on `Q, Qv, A, betas`, interchanging the parallel loop inwards and distributing it over the sub-statements gives us a pseudocode representation of the kernels that we need:

```
for (int kk = 0; kk < n_cols; kk++) { // parallel
    Q[kk] = identity(m);
}
for (int k = 0; k < n; k++) { // sequential
    for (int kk = 0; kk < n_cols; kk++) // parallel kernel
        v[kk] = A[kk][k:m,k];
    for (int kk = 0; kk < n_cols; kk++) // parallel batched matrix-vector
    mult
        Qv[kk] =  Q[kk][:m, k:m] * v[kk];
    for (int kk = 0; kk < n_cols; kk++) // parallel kernel
        Qv[kk] = betas[kk][k] * Qv[kk];
```

```
11      for (int kk = 0; kk < n_cols; kk++) // parallel batched rank-1 update
12          Q[kk][:m, k:m] = Q[kk][:m, k:m] - Qv[kk] * v[kk]^T;
13      }
14  }
```

The code for setting the householder vector `v[kk] = A[kk][k:m,k];` looks like this:

```
1   __global__ void set_vs_initial(double* vs, int k_to_m, double* Ahats, int
        A_size, int* n2s, int maxn1, int k, int n)
2   {
3       const unsigned int tid = threadIdx.x + blockIdx.x * blockDim.x;
4       const unsigned int kk = tid / k_to_m;
5       const unsigned int i = tid % k_to_m;
6       if (tid < n)
7       {
8           if (n2s[kk] - 1 < k)
9               vs[k_to_m * kk + i] = 0;
10          else
11          {
12              if (i == 0)
13                  vs[k_to_m * kk + i] = 1;
14              else
15                  vs[kk * k_to_m + i] = Ahats[A_size * kk + IDX2C(k + i, k,
        maxn1)];
16          }
17      }
18  }
```

Since the innermost dimension of `v` and `Ahats` are accessed by consecutive threads, we have coalesced access. The kernel for the scalar-vector multiplication is similarly simple and has coalesced access. For the rank-1 update, I use the batched matrix-matrix multiplication from CUBLAS, which updates a matrix C with a the result of a matrix-matrix multiplication. In this case, my matrices are $\underbrace{Qv}_{m\times 1}$ and $\underbrace{v^T}_{1\times k-m}$, which makes the total operation a rank-1 update.

3.2.4 Calculating $\tilde{J}$

Grote and Huckles way of calculating $\tilde{J}_k$ for column k involves first setting $\tilde{J}_k$ to be the set of all new column indices of A that appear in all $L_k = I_k \cup \{k\}$ rows but not in J_k and then eliminating non-profitable indices from that set. The size of this set can grow to be very large. When allocating space for sets that we do not know the size of beforehand, we have to overestimate the amount of memory needed, and we would normally allocate memory

for $|L_k| \cdot \max_j\{nnz(A(:,j)\}$ integers. My alternative approach uses min-heaps to calculate $\tilde{J}$. Each column k requires only to allocate memory for $3 \cdot |L_k|$ integers which is a significant reduction in the constant factor of the allocations needed.

For a given column, the min heap is initialized with the first column index of each row in $l \in L$. It also stores how many nonzero elements are left in this row along with a pointer to the next element. While the heap is non-empty, the smallest element is popped. If it is different from the previous examined index, and it is not contained in J, we compute ρ_j^2 and put $(-\rho_j^2, j)$ into another min-heap with a capacity of 5, which in turn keeps track of the 5 smallest ρ_j^2 by popping the smallest $(-\rho_j^2, j)$ in each iteration. Finally of the 5 smallest (ρ_j^2, j) pairs, we keep the indices j, where $\rho_j^2 \leq \text{avg}_j\{\rho_j^2\}$

I use the same strategy for the calculation of I and $\tilde{I}$, since it is essentially a union of sorted vectors. The step, where I check, whether a new index is contained in the previous set, requires that I have the sorted representation of the set, which is maintained using CUB's segmented radix sort [6].

This part of the implementation is not currently performed on the GPU, however it should be relatively simple to implement, since the min heap is just an array with an associated set of functions. Allocating all the heaps in a single array on the GPU and adjusting the functions to take a column number and a max heap size and using `max_heap_size * k` as the starting index for each heap should be enough for an embarrassingly parallel solution.

3.2.5 Post-SPAI Assembly of M

After the while loop has finished, we can assemble the result in CSC format. Since each J_k is not sorted, we do this by performing a segmented key-value sort of `Js` $= \{J_0, J_1, J_2, \ldots J_{n-1},\}$ and `mtildes` $= \{\tilde{m}_0, \tilde{m}_1, \tilde{m}_2, \ldots \tilde{m}_{n-1},\}$. With the offsets being the result of an exclusive scan of the array of n_2s. This can easily be done using CUB's `cub::DeviceScan::ExclusiveSum` and `cub::DeviceSegmentedRadixSort::SortPairs`. The sorted values become the values $Mval$ of the CSC representation, the J indices become the row indices $Mrowinds$, and the result of the scan becomes the column pointer $Mptr$.

3.3 Edge Cases

When there is a matrix, A, with an empty column, I is empty initially, and we cannot compute a QR-decomposition, since $A(I, J)$ is also empty. This means that we have to handle the case where $n_1 = 0$ in the while loop. Normally we would insert $A(I, \tilde{J})$ at position $(0, n_2)$ in the new A and $A(\tilde{I}, \tilde{J})$ at position (n_1, n_2), however since I is empty, we have to insert it at $(0, 0)$.

I have also found that matrices with empty columns produce singular R's, which means that there is no solution to the best approximation of $M(:, k)$ for the given sparsity structure. I have not found a way to handle this gracefully.

3.4 Discussion

3.4.1 Improvements

There are a number of ways to improve the parallel implementation. One is to realize that the upper bound for $|I_k|$ is $\max_k\{|J_k|\} \cdot \max_j\{nnz(A(:, j))\}$. This would allow us to allocate memory for all matrices and arrays beforehand with $J_{max} = \max_k\{|J_k|\}$ and $I_{max} = J_{max} \cdot \max_j\{nnz(A(:, j))\}$. The sizes of the arrays would be:

$J : n \times J_{max}$
$I : n \times I_{max}$
$\tilde{A} : n \times I_{max} \times J_{max}$
$Q : n \times I_{max} \times I_{max}$
$\tilde{m}_k : n \times J_{max}$
$\tilde{r} : n \times I_{max}$

Reducing the amount of GPU allocations would greatly increase the potential performance of the algorithm. I also ran into issues with being out of memory on larger matrices, which I have not been able to investigate due to time restrictions but it may have something to do with the fact that I double my allocation size, whenever I need more memory.

Another strategy for increasing the performance is to include some sort of caching strategy for the QR-decompositions. When we have to decompose a matrix, that has already been decomposed, we could fetch the result from a cache instead of recomputing the result. Most matrices have a lot of overlap between the columns, and from [15] Table 4.4, we see that between 2 and 99% of QR-decompositions are made redundant by using a caching strategy, depending on the matrix.

3.4.2 Alternatives

The construction of I and $\tilde{I}$ could be performed using a combination of the parallel segmented basic blocks. Since they are essentially a segmented union of sorted arrays, they can be performed in the following way with I as an example. Each column k represents a segment, each of which contains the row indices of $rowinds(A[:, J]) = rowinds(A[:, j_0, j_1, \ldots]) = [\underbrace{i_0^0, i_1^0, \ldots}_{A[:,j_0]}, \underbrace{i_0^1, i_1^1, \ldots}_{A[:,j_1]}, \ldots]$. Sorting each this for each column k involves a segmented sort. You could them map a function that marks unique elements with a 1. Performing a segmented scan with the addition operator on this array would produce the indexes that each element should be scattered copied to. This method of stream compaction is described in section 39.3.1 of GPU Gems 3 [16].

The parallel basic blocks of map and scan used for the operations above, are provided in Futhark [3] [2]. This language provides allows the user to write functional, nested parallel programs written in terms of map, reduce, scan etc. It also allows compilation to CUDA code, which means that you could write the code in Futhark and link it to your CUDA application, where you perform the rest of the algorithm.

Finding the 5 smallest ρ_j^2 in the calculation of $\tilde{J}$ could also be written in terms of a map reduce. Mapping J_{cand} with a function that produces a 5-tuple containing, where each entry contains a (ρ_j, j) pair and reducing with an operator that that maintains the 5 pairs with the smallest ρ_js would produce the desired result.

4 Evaluation

4.1 Sequential Implementation

A large concern of this project was implementing the algorithm correctly according to the literature. Since I am not able to compare to other implementations, the second best measure of the quality of my implementation, would be to see how well it performs as a preconditioner. I could do this by using it to precondition a linear system and solving it using an iterative solver such as GMRES, BCG, BI-CGSTAB and CG. This is what Grote and Huckle do and most other papers on SPAI variations. However, since I am limited on

time, I will instead mainly be evaluating my implementation based on the two measurements, that are also often reported in the literature:

(1) The Frobenius norm of the residual, $\| AM - I \|_F$, as a measure of the proximity of AM to the identity. This is what the algorithm minimizes for a given sparsity structure and is a good proxy measure for the quality of the preconditioner.
(2) The ratio of the number of non-zeros M to the number of non-zeros in A: $\frac{nnz(M)}{nnz(A)}$

I will also gague the effectiveness of the algorithm based on whether or not each column converges to a satisfactory error within the specified number of iterations and how quickly. However, I cannot compare against other implementations as I have not found this measure reported in the literature.

4.1.1 Correctness

I will almost exclusively be comparing the results of my sequential implementation against the results reported by Grote and Huckle. This is due to the fact that my partially parallel implementation runs for the full number of iterations because to my naive stopping criteria. Grote and Huckle do not report their max number of iterations, but I have kept mine at 10. The max fill-in per iteration is 5.

Table 1 shows my a reproduction of Table 2 from Grote and Huckle, page 14 [1] using results from my sequential implementation. The table contains the Frobenius norm of the residual and the ratio of non-zero values in M to A after the SPAI algorithm using different stopping criteria, ε, for the orsirr_2 matrix: an oil resevoir simulation matrix with $n = 886$ and $nnz = 5970$.

	$\| AM - I \|_F$	$\frac{nnz(M)}{nnz(A)}$
$M = I$	1.54×10^6	0.148
$\varepsilon = 0.6$	14.27	0.320
$\varepsilon = 0.5$	11.30	0.607
$\varepsilon = 0.4$	8.977	0.891
$\varepsilon = 0.3$	7.131	1.528
$\varepsilon = 0.2$	4.987	3.144

Table 1: $\| AM - I \|_F$ and ratio of non-zero values in M to A for different values of ε for the orsirr_2 matrix

Cross-referencing the results, I get exactly the same norm and ratio from $\varepsilon = 0.6$ until $\varepsilon = 0.4$. For $\varepsilon = 0.3$ and $\varepsilon = 0.2$, my norm is slightly higher than theirs, but my ratio of non-zeros is lower. Since their results have higher fill-in it indicates to me that they have used a higher number of maximum iterations. If I use a maximal number of iterations of 20, I get the same result of $\| AM - I \|_F = 4.817$ and $\frac{nnz(M)}{nnz(A)} = 3.393$ for $\varepsilon = 0.2$.

The rest of the results reported from Grote and Huckle are only on the ratio of non-zeros for different ε, iterations and matrices. I will evaluate my results against theirs for the shermanx black oil simulators, consisting of 5 matrices:

- **sherman1**: $n = 1000$, $nnz = 3750$,
- **sherman2**: $n = 1080$, $nnz = 23094$
- **sherman3**: $n = 5005$, $nnz = 20033$
- **sherman4**: $n = 1104$, $nnz = 3786$
- **sherman5**: $n = 3312$, $nnz = 20793$

My results are reported in Table 2. They show a great reduction in the Frobenius norm along with a set of relatively small ratios of non-zeros in M compared to A. Their results match mine closely except for sherman2 and sherman5, where I have slightly higher ratios of nonzeros. This could be due to the fact that my max. $nnz(m_k)$ is 101, and 51, where theirs are 100 and 50. This is due to the fact that my $nnz(m_k) = 1 + \text{maxiter} \times \max\{|\tilde{J}|\} = 1 + 10 \times 5$ and $1 + 10 \times 10$ respectively. Overall, the results indicate to me that my sequential implementation is correct.

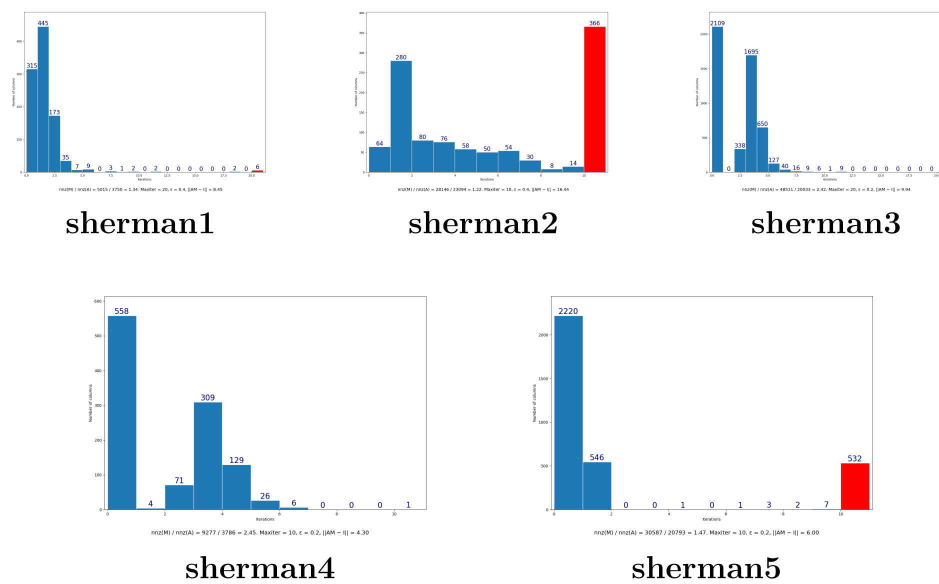
	$\| AM - I \|_F$ before	$\| AM - I \|_F$ after	$\frac{nnz(M)}{nnz(A)}$
sherman1, $\varepsilon = 0.4$, max iter = 20	68.017	8.454	1.337
sherman2, $\varepsilon = 0.4$, max iter = 10	7.004×10^9	16.442	1.219
sherman3, $\varepsilon = 0.2$, max iter = 20	1.357×10^7	9.941	2.421
sherman4, $\varepsilon = 0.2$, max iter = 10	483.988	4.304	2.450
sherman5, $\varepsilon = 0.2$, max iter = 10	1.4032×10^4	5.996	1.471

Table 2: $\| AM - I \|_F$ and ratio of non-zero values in M to A for the shermanx matrices.

4.1.2 Convergence

To show the fact that most columns converge to their desired stopping criteria and do so at a reasonable speed, I have compiled a series of histograms of

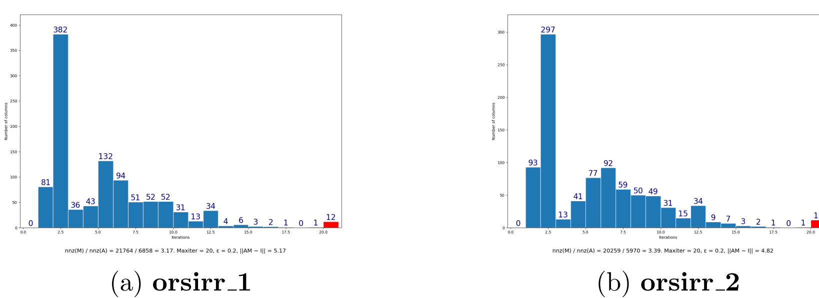
how many iterations it takes to converge, x-axis, for how many columns y-axis. The last bucket of each histogram shows the amount of columns that did not converge within the set number of iterations and is marked in red in each figure. I performed the test on the shermanx matrices with the same stopping criteria from Table 2 shown in Figure 1 and the and orsirr_1 and orsirr_2 matrices with $maxiter = 20$ and $\varepsilon = 0.2$ shown in Figure 2. Larger figures are included in the appendix.



sherman1 **sherman2** **sherman3**

sherman4 **sherman5**

Figure 1: Convergence results from shermanx matrices



(a) **orsirr_1** (b) **orsirr_2**

Figure 2: Convergence results from orsirr_x matrices

We see that there is a great deal of variation of the convergence rates between the matrices. **sherman2** has a lot of columns that do not converge which is also clear in Table 2. The rest of the matrices have more satisfactory convergence rates.

4.2 Parallel Implementation

4.2.1 Correctness

During the development of the CUDA implementation, I ensured that it matched the exact results from the sequential Python implementation. However, as mentioned, the parallel implementation runs for the full number of iterations for every column because of my naive stopping criteria. In order to compare the implementations, I let the Python version run for the full number of iterations for every column as well. The results of the fill-in, $\frac{nnz(M)}{nnz(A)}$, are exactly the same between the two implementations, Indicating to me that the sparsity pattern update is correct in my parallel implementation. However the Frobenius norm is 3.856 for my parallel implementation and 3.728 for my Python implementation when running 10 iterations for all columns on the orsirr_2 matrix. This may be due to an error of my parallel implementation. The discrepancy could also arise from differences in the QR-decomposition and upper-triangular system solving functionality from numpy and scipy versus CUBLAS.

5 Further Work

During this project, I did not finish implementing the parallel version of the algorithm in full. This presents a great opportunity for further work. My section on how to parallelize the remainder of the implementation also has room for improvement, as I did not get specific with code examples.

Additionally, my evaluation can not be considered satisfactory for evaluating the performance of the SPAI algorithm. Further work could include a full evaluation of the performance of my preconditioner in helping the convergence rates of iterative solvers. Performing a full evaluation by including the time taken to compute the preconditioner in a fully parallel implementation, would reveal whether or not, calculating the preconditioner is worth the effort.

6 Conclusion

In this project I have presented the dynamic SPAI algorithm with a focus on understanding the algorithm in detail in terms of what is required for a correct and efficient implementation. With this understanding, I have implemented a sequential Python and a partially GPU-parallel CUDA version of the algorithm and given some possible methods for parallelizing the remaining sequential parts. Experimental results validate the correctness of the both implementations in terms of the their ability to minimize the Frobenius norm of the residual for a given sparsity structure.

References

[1] Marcus Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18:838–853, 01 1996.

[2] R. Schenck, O. Rønning, T. Henriksen, and C. E. Oancea. Ad for an array language with nested parallelism. In *2022 SC22: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (SC)*, pages 829–843, Los Alamitos, CA, USA, nov 2022. IEEE Computer Society.

[3] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional gpu-programming with nested parallelism and in-place array updates. *SIGPLAN Not.*, 52(6):556–571, jun 2017.

[4] NVIDIA®. Cuda toolkit documentation 12.1 update 1. https://docs.nvidia.com/cuda/index.html#cuda-toolkit-documentation-v12-1, 2023.

[5] NVIDIA®. cublas documentation, v.12.1. https://docs.nvidia.com/cuda/cublas/index.html#, 2023.

[6] NVIDIA®. Cub documentation. https://nvlabs.github.io/cub/, 2022.

[7] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[8] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[9] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[10] Matous Sedlacek. Sparse approximate inverses for preconditioning, smoothing, and regularization. Technische Universität München, 2012.

[11] Andreas Roy. Untersuchung dünnbesetzter qr-verfahrenbei der berechnung dïnnbesetzter approximativer inverser. Technische Universität München, 2008.

[12] Robert Andrew and Nicholas Dingle. Implementing qr factorization updating algorithms on gpus. *Parallel Computing*, 40(7):161–172, 2014. 7th Workshop on Parallel Matrix Algorithms and Applications.

[13] Cosmin E. Oancea. *Lecture Notes for the Software Track of the PMPH Course*. sep 2018.

[14] Andrew Kerr, Dan Campbell, and Mark Richards. Qr decomposition on gpus. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-2, page 71–78, New York, NY, USA, 2009. Association for Computing Machinery.

[15] Alexander Kallischko. Modified sparse approximate inverses (mspai) for parallel preconditioning. Technische Universität München, Zentrum Mathematik, 2008.

[16] Hubert Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

A Appendix

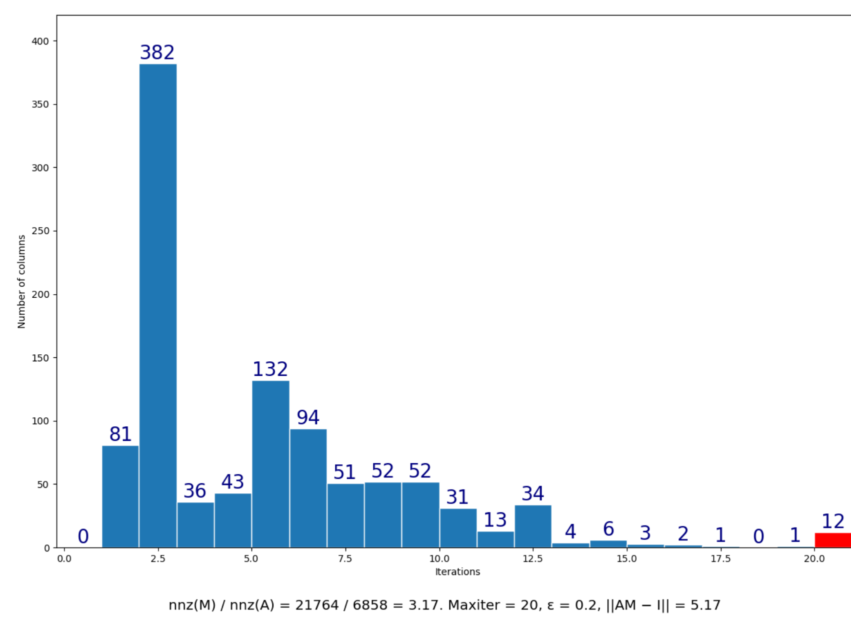


Figure 3: **orsirr_1**

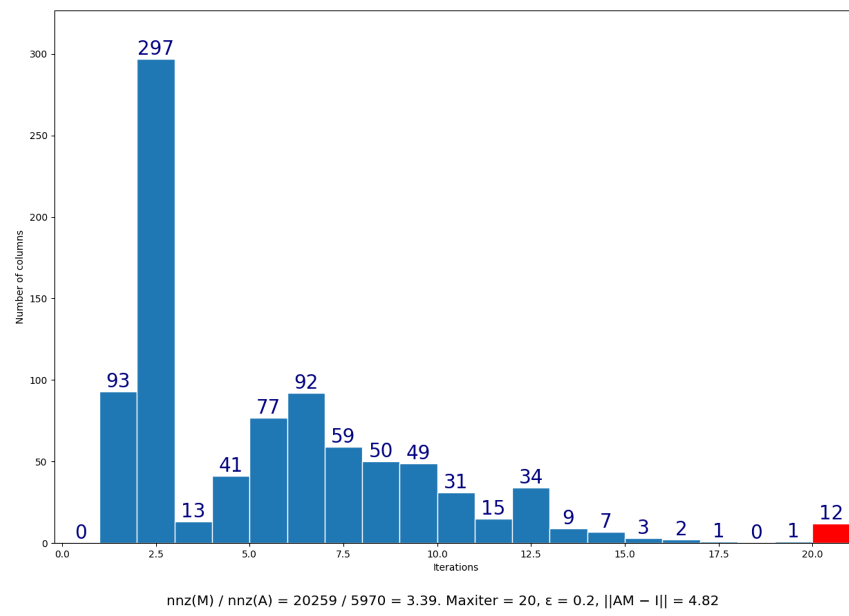


Figure 4: **orsirr_2**

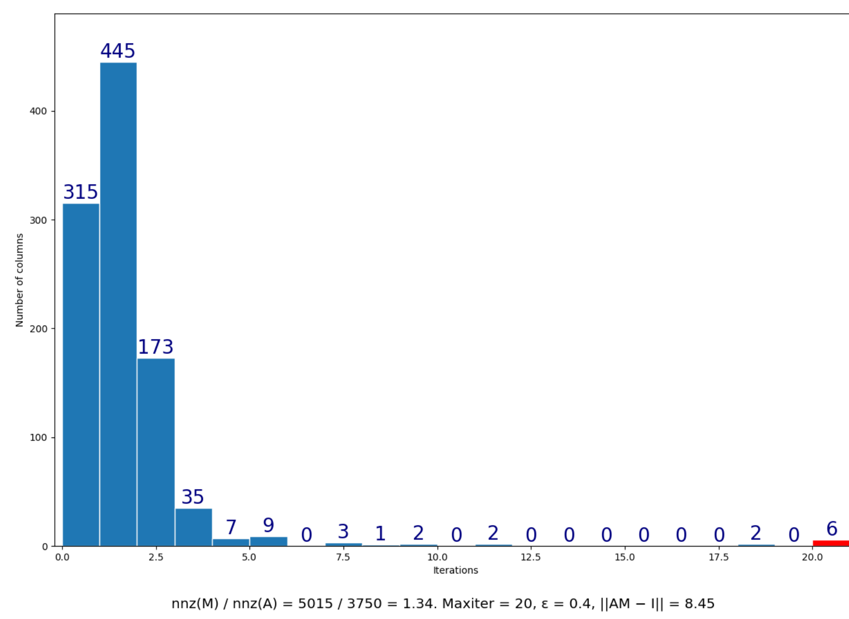


Figure 5: **sherman1**

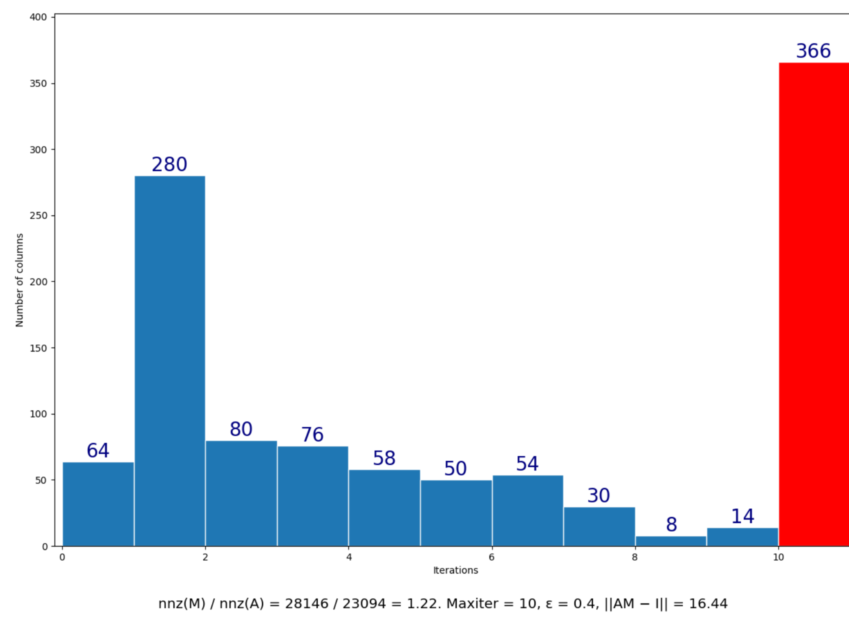


Figure 6: **sherman2**

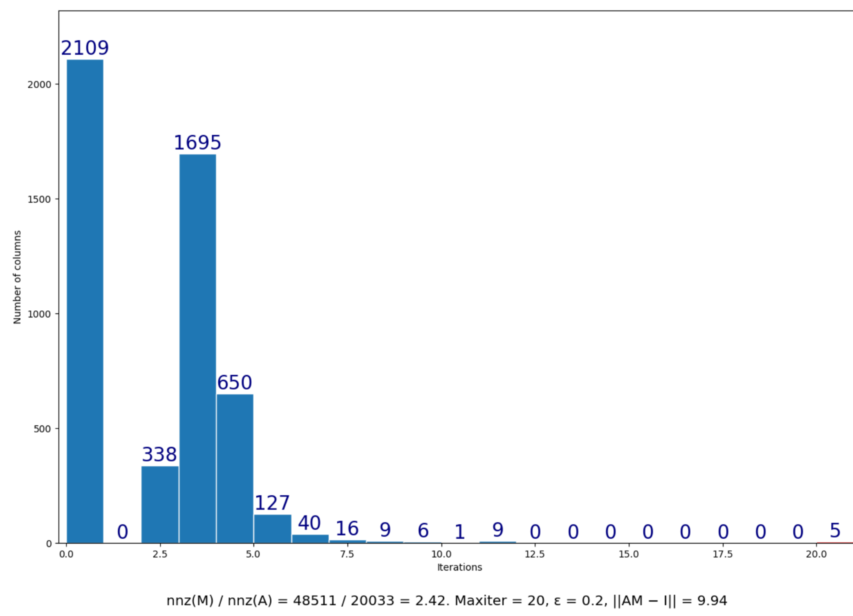


Figure 7: **sherman3**

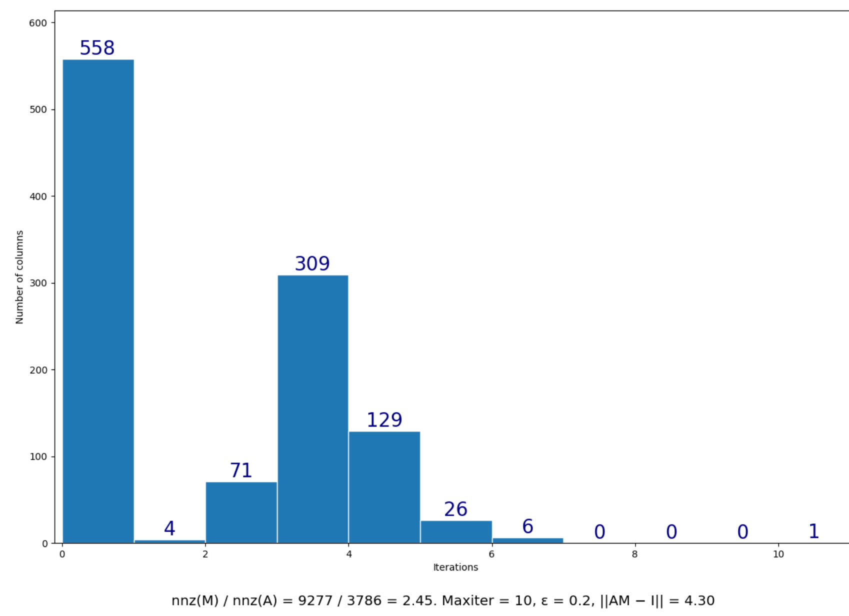


Figure 8: **sherman4**

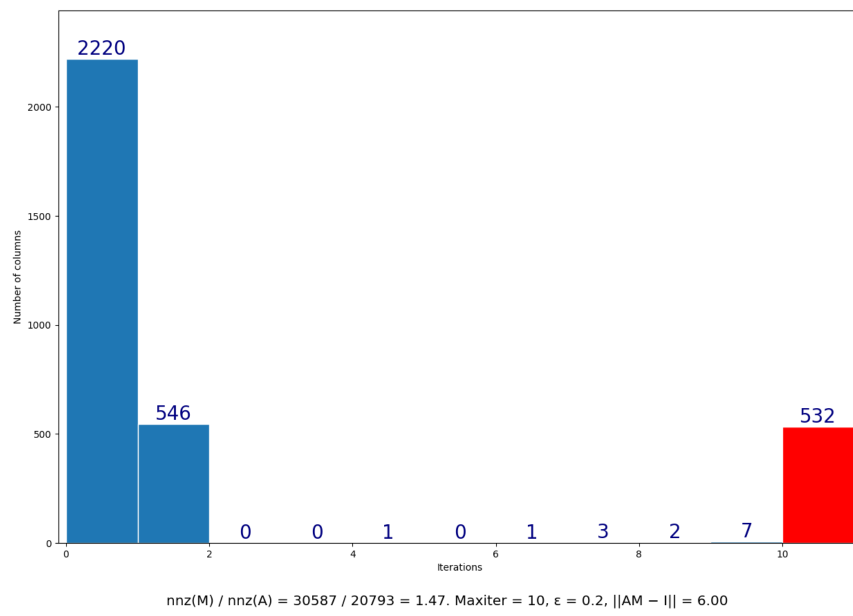


Figure 9: **sherman5**