KØBENHAVNS UNIVERSITET

Parallel Implementation of the SPAI algorithm

Caroline Amalie Kierkegaard (qlj556) and Mikkel Willén (bmq419)
Supervisor: Cosmin Oancea

June 5, 2023

Abstract

This thesis explores the Sparse Approximate Inverse (SPAI) preconditioner, which calculates an approximate inverse of a large and sparse matrix. The SPAI algorithm iteratively constructs a sparse approximation of an inverse matrix by minimising the norm, column by column, while preserving the sparsity. It is inherently parallel and thus our aim is to execute it efficiently on the Graphics Processing Unit (GPU). The thesis elaborates on the original SPAI algorithm by Grote and Huckle by incorporating theoretical explanations of QR decomposition with Householder reflections and permutations. We have implemented sequential prototypes in Python and C, followed by a parallel version using CUDA kernels. Experimental results demonstrate the accuracy of the SPAI algorithm. We compare the sequential SPAI implementations to Scipy and cuSOLVERs functions for finding the exact inverse and find the run time of the library functions superior. We perform experiments on the CUDA kernels implemented in the parallel SPAI implementation. The results show that the kernels executed on the GPU exhibits superior run time to the corresponding sequential code running on the Central Processing Unit (CPU).

This thesis shows that the SPAI preconditioner efficiently computes approximate inverses of e.g. large Hessian matrices in the Newton method. We show that the parallel implementation running on the GPU outperforms the sequential implementations running on the CPU.

Contents

1 Introduction

For centuries mathematicians have been faced with the challenge of solving systems of linear equations, and without calculators and computers, solving larger systems proved difficult. Directly finding the exact inverse A^{-1} of a large and sparse matrix A would historically be infeasible, and nowadays extremely expensive in terms of computational work and memory. In this thesis we will consider an iterative method for constructing a sparse approximation to the inverse of such a matrix A, where the sparsity is preserved in the inverse. Due to the preservation of the sparsity pattern, the work will be proportional with the amount of nonzero elements and not the size of the matrix.

Since the 1970's, the technique of preconditioning have been investigated and lead to great results. Among these are Grote and Huckle's sparse approximate inverse preconditioner (SPAI) based on the Frobenius norm minimisation [1]. SPAI is an effective preconditioner, because it is able to update a given sparsity structure automatically, while maintaining the sparsity of the original matrix for each iteration. Thus it splits up each computation into an independent least squares problem for each column and is inherently parallel. Inherently parallel preconditioners become of increasing importance as parallel clusters with millions of cores are build and the paradigm of parallel programming develops [3]. A great advantage is the time and space complexity, which, for a sparse matrix, relies on the number of nonzero elements rather than the size of matrix.

The ongoing efforts to advance the SPAI algorithm stem from its practical applications on large amounts of sparse data. Determining the optimal solutions inexpensively with the Newton method is essential in different areas such as statistics, applied mathematics, numerical analysis, economics and finance [5]. The Newton method finds the roots of a differential function F, which are the solutions to the equation $F = 0$. In order to do so, we have to find the inverse of a Hessian matrix, which is a square matrix of second-order partial derivatives. This is were the SPAI preconditioner becomes highly relevant, because computing the exact inverse of a Hessian of large dimensions will be require many resources.

To find the exact inverse of a dense matrix, we perform Gaussian elimination on the augmented A and identity matrix I [6].

$$AA^{-1} = I$$
$$[A|I] \rightsquigarrow [I|A^{-1}]$$

Since Gaussian elimination will be extremely laborious for a large, sparse matrix, the method of finding the approximate inverse is very relevant.

Developing on the initial SPAI algorithm by Grote and Huckle, several theses have been dedicated to improving the Sparse Approximate Inverse preconditioner.

Among these are the dissertation *Modified Sparse Approximate Inverses (MSPAI) for Parallel Preconditioning* by Kallischko, which introduces variants of the Sparse Approximate Inverse technique; the modified SPAI (MSPAI) algorithm and the factorised SPAI (FSPAI) algorithm [2]. The dissertation *Sparse Approximate Inverses for Preconditioning, Smoothing, and Regularization* by Sedlacek improves on the SPAI algorithm and its variants also by modifying and factorising it and goes into depth with the theory behind updating the QR decomposition [3]. Gao, Chu and Wang presents a new heuristic SPAI preconditioner in the the paper *HeuriSPAI: a heuristic sparse approximate inverse preconditioning algorithm on GPU*, which combines the advantages of static and dynamic SPAI algorithms [8]. The three papers each present relevant solutions to problems we have encountered in our implementation, which we will discuss further in the related works part of the discussion.

In this thesis we have implemented the SPAI algorithm both sequentially and parallel. We have implemented a highlevel version in Python, which we use to test against. We use the python imple-

mentation to debug the C and CUDA versions and later compare the accuracy and the runtime of the implementations. We take a deep dive into the SPAI algorithm and explain parts of the algorithm that are not exhausted in Grote and Huckles [1] explanation of the algorithm. This includes the use of QR decomposition with Householder reflections and how to update the QR decomposition efficiently.

Section 2 will provide the background for the work in the thesis. We will account for preconditioning and explain the Compressed Sparse Column (CSC) format used in the implementation.

Section 3 will provide the theoretical prerequisites for the SPAI algorithm and fulfill the theoretical learning goals. We will explain QR decomposition and its relevance for the algorithm. We will elaborate further on the method for computing the QR decomposition using Householder reflections. We will explain a technique for how to find the exact inverse of a matrix and account for row and column permutation matrices.

Section 4 is devoted to the sparse approximate preconditioner (SPAI) by Grote and Huckle and will elaborate on sparse approximate inverses and the algorithm for determining such. We will methodically go trough the algorithm and explain in depth the most relevant parts. We will explain the Frobenius norm minimisation, how to compute the sparsity structure $\mathcal{J}$ and $\mathcal{I}$ and how to apply the Householder QR decomposition. We will describe how to improve upon M by updating the sparsity structure and the QR decomposition. We will present our pseudocode, that we use as a foundation for our implementation. We will go trough a numerical example for which, we have traced by hand every step of one iteration of the algorithm for one subproblem.

Section 5 concerns our implementations of the SPAI algorithm. We have implemented a sequential prototype in Python to use as a basis for our tests. In C we have implemented the algorithm sequentially and used cuBLAS functions to compute the QR decomposition and the inverse of R. Finally we have implemented the algorithm parallel in CUDA. The implementation does not work sufficiently for larger matrices, but we will describe and explain how we have implemented a parallel implementation aimed at GPU execution. We will thoroughly explain the decisions we have made and discuss the shortcomings present in our implementations. Additionally, we will present relevant code snippets to provide a clearer understanding of our approach.

Section 6 shows the test results of our experiments. We have tested the accuracy of our implementations of the SPAI algorithm. We have tested our implementations on different sizes and sparsity degrees of randomised matrices. We have performed experiments to find the run times of our sequential SPAI implementation and Scipy and cuSOLVERs library functions for finding the exact inverse. We have tested the run time and GB/s for individual CUDA kernels from our parallel implementation and compared with corresponding sequential code.

Section 7 compares and analyses the results obtained in numerical experiments. We will discuss the accuracy of our Python, C and CUDA implementations. The result shows that the SPAI algorithm successfully converges when reaching the accepted tolerance. We have compared the run times of our sequential SPAI implementation to library functions for finding the exact inverse. The results show that the library functions outperform the sequential implementations. In theory the parallel implementation should exhibit better run times for larger matrices, but since the implementation does not work sufficiently for large matrices, we cannot show this. We have tested individual CUDA kernels from our parallel implementation and the results clearly demonstrate that the GPU-executed kernels exhibit significantly faster run times compared to their corresponding sequential code running on the CPU. For a broader perspective, we discuss the advantages of parallelism. This section will also discuss how related works have contributed to our thesis and developed further on the SPAI algorithm.

Section 8 is the conclusion.

The link to our GitHub repository is:
`https://github.com/mikkelwillen/SPAI`.

This code presented in this thesis is based on the commit `"Commit presented in the thesis"` from June 5, 2023. Changes and optimisation may have been committed since then.

2 Background

This section will provide the background for the work in order to make the thesis easier to follow. We will account for preconditioning and explain the Compressed Sparse Column (CSC) format used in the implementation.

2.1 Preconditioning

Preconditioning is a technique used for enhancing convergence speed. Many iterative methods converge very slow for not preconditioned systems. The goal of preconditioning is to modify the system in such a way that an iterative method converges significantly faster.

In linear algebra a preconditioner P of a matrix A is a matrix such that $P^{-1}A$ has a smaller condition number than A [6]. The condition number measures how much the output value can change for a small change in the input argument. So it is used to measure how much an error in the output results from an error in the input.

Preconditioners play a valuable role in iterative methods used to solve linear systems of the form $Ax = b$. By applying a preconditioner, the convergence rate of most iterative linear solvers improves. This improvement occurs due to the reduction in the condition number of the matrix, which is a direct outcome of preconditioning. Instead of solving the original linear system $Ax = b$ for x, one may consider the right preconditioned system

$$AP^{-1}(Px) = b$$

and solve

$$AP^{-1}y = b$$

for y and

$$Px = y$$

for x. Alternatively, one may solve the left preconditioned system

$$P^{-1}(Ax - b) = 0.$$

Assuming the preconditioner matrix P is nonsingular, both systems yield an identical solution to the original system. The left preconditioning approach is used more often.

The Sparse Approximate Inverse preconditioner minimises $\|AM - I\|_F$, where $\|\cdot\|_F$ is the Frobenius norm and $M = P^{-1}$.

The concept of preconditioning is theoretically straightforward, but practical implementations require careful consideration of various technical details. When constructing a preconditioner M, we must find a balance between reducing the number of iterations required and not making the preconditioner too computationally expensive to construct. A well chosen preconditioner must be

- relatively cheap to construct
- not use too much memory
- improve the condition number of A
- be parallelisable when both constructed and implemented
- and performing linear solves with the preconditioner should be cheap [4]

2.2 Sparse matrices

Matrices can have specific structures or hold special properties. Some matrices only have a small amount of nonzeros compared to their dimensions, and these are called sparse matrices. There is not a strict definition of a sparse matrix, but a general perception is that a matrix will be called sparse if the storage of only the nonzero elements and/or the application of an algorithm adapted to sparse matrices lead to a computational advantages over a dense matrix [2].

Sparse matrices enables us to construct methods which profits from the sparse structure in terms of fast algorithms and efficient storage.

2.3 Sparse Compressed Column format

When storing a sparse matrix, we do not want to waste space storing the great amount of zeros. We can store the nonzeros either row- or column-wise. Since the SPAI algorithm is computed column-wise, we want fast access to the columns and therefore we would like to store our data in a column format. The format we will use in this thesis is called Sparse Compressed Columnn (CSC) format.

In the format, the matrix is stored in three arrays. The first array `offset` contains the accumulated number of nonzeros elements in each column. The `flatData` contains the nonzero entries ordered after columns. The `flatRowIndex` contains the row indices of the nonzero entries also ordered after columns. Here is an example of the CSC storage format:

$$A = \begin{pmatrix} 20 & 0 & 0 \\ 0 & 30 & 10 \\ 0 & 0 & 0 \\ 0 & 40 & 0 \end{pmatrix} \xrightarrow{\text{CSC format}} \begin{array}{lcl} \texttt{offset} & = & \{0,1,3,4\} \\ \texttt{flatData} & = & \{20,30,40,10\} \\ \texttt{flatRowIndex} & = & \{0,1,3,1\} \end{array}$$

Figure 1: Example of the CSC storage format

The compressed storage saves communication between processing elements i parallel environments, because we only have to exchange the nonzero elements [3].

3 Theoretical prerequisites for the SPAI algorithm

This section will provide the theoretical basis for the sparse approximate inverse preconditioner and the implementation of it.

3.1 QR decomposition

QR decomposition is a decomposition of a matrix A into a product $A = QR$, consisting of an orthonormal matrix Q and an upper triangular matrix R. QR decomposition is often used to solve the linear least squares problem and that is why it is relevant later in the SPAI algorithm.
There are several ways of computing the QR decomposition of a matrix, but in this thesis we will solely focus on the method using Householder reflections. The CUBLAS library, we will use for the implementation of the SPAI algorithm in C and CUDA, computes the QR decomposition via Householder reflections.

3.2 QR decomposition with Householder reflections

QR decomposition with Householder reflections computes the upper triangular matrix R from a matrix A of size $m \times n$ by applying a sequence of Householder reflections to A. A matrix H of the form

$$H = I - 2\frac{vv^T}{v^T v} \tag{1}$$

is called a Householder reflection, with v being a Householder vector and I being the $m \times m$ identity matrix equal to $Q^T Q$. We can define the first Householder transformation as the matrix H_1 from the first Householder vector v_1 :

$$H_1 = I - 2 \cdot \frac{v_1 \cdot v_1^T}{v_1^T \cdot v_1} \tag{2}$$

v_1 is the first Householder vector for the first column vector a_1 from our input matrix A. We can calculate v_1 from:

$$v_1 = a_1 + \operatorname{sign}(a_{11}) \|a_1\| e_1,$$

with $\|a_1\|$ being the vector length of the first matrix column in A and and e_1 being the first column of the identity matrix. The sign function is an extraction of the sign of a number [7].

This way, we don't compute our Q explicitly but instead as a sum of Householder reflections.

$$Q = H_1 \cdots H_k, \tag{3}$$

where $k = \min(m-1, n)$ and H is the Householder reflections. We can find R by triangularising A with the Householder reflections.

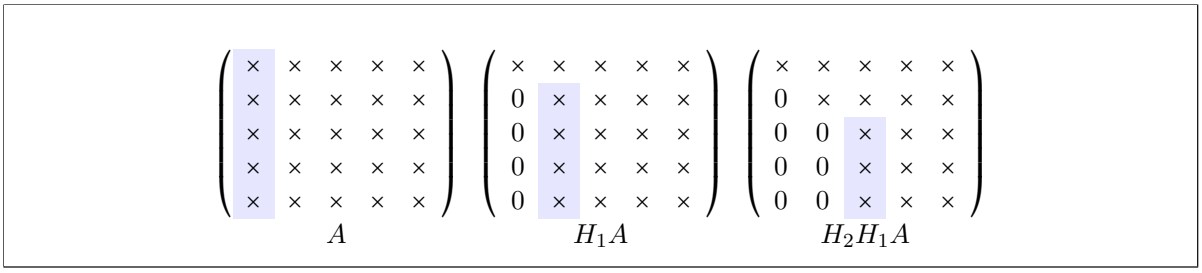

Figure 2: Triangularising A with Householder reflections

Figure 2 illustrates how a sequence of Householder transforms may be chosen from columns of A to bring it into triangular form. Once the input matrix has been overwritten with the first Householder transformation H_1A, the first column will contain zeros below the diagonal. We proceed to the next column and move one row down to repeat the process, calculating the next Householder transformation H_2. This process is iterated for all columns in the updated input matrix corresponding to R, continuing until A is triangularised [13].

$$R = H_k \cdots H_1 A \tag{4}$$

Since we can reconstruct the Householder transformations H_k quite easily from the Householder vectors, we can store them efficiently in the lower triangular part of R, that is not used for anything else than zeros.

Kerr, Campbell and Richards presents an algorithm for computing the QR decomposition of A with Householder reflections [13]. Here is our version of it:

Algorithm 1: Compute the QR decomposition of A via Householder reflections

$Q \leftarrow I$
for $k = 1$ *to* n **do**
  $Q(1:m, k:m) = Q(1:m, k:m) - \beta Q(1:m, k:m) v v^T$
end for

Here v is the Householder vector and $\beta = \frac{2}{v^H v}$ [13].

We can observe from the analysis of the algorithm's running time that it operates in $O(n^3)$ complexity. When examining the section inside the for loop, we notice that the algorithm performs two matrix-vector multiplications and a matrix-matrix subtraction. Matrix-vector multiplication takes $O(m \cdot m)$ time since each element in the rows is multiplied by the corresponding element in the vector. This results in one multiplication per element in the $m \times m$ sized matrix. The matrix-matrix subtraction, similarly, involves subtracting each element in the first matrix from the corresponding element in the second matrix, leading to a time complexity of $O(m \cdot m)$. As the for loop iterates n times, the overall running time of the algorithm becomes $O(m \cdot n^2)$. In the case of SPAI, where Q is a square matrix, the running time of QR decomposition with Householder reflections is $O(n^3)$, where $n \times n$ represents the size of the matrix Q.

3.3 Inversion

A $n \times n$ matrix A is invertible if there exists a matrix B such that:

$$AB = BA = I_n \tag{5}$$

B is the inverse of A, which is commonly denoted as A^{-1} [6]. We can compute the matrix inverse by solving the linear system $AB = I$, for example:

$$\left(\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right) \xrightarrow{Gauss-Jordan} \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & b_{11} & b_{12} & b_{13} \\ 0 & 1 & 0 & b_{21} & b_{22} & b_{23} \\ 0 & 0 & 1 & b_{31} & b_{32} & b_{33} \end{array}\right) \tag{6}$$

Then B will be the inverse. We compute the left side of the arrow from the right side by performing Gauss-Jordan elimination.

Gauss-Jordan elimination is the process of performing a series of elementary row operations to produce a matrix that is on reduced row-echelon form. This can be performed by ensuring that each pivot has the value 1 and eliminating values above and below the pivots [6].

3.4 Row and column permutations

A permutation matrix is a square matrix derived from the identity matrix of the same size through a rearrangement of its rows. The resulting matrix remains row equivalent to an identity matrix [6]. In a permutation matrix, each row and each column contains precisely one non-zero entry, which is always a 1. Consequently, there exist two 2×2 permutation matrices.

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{7}$$

There are $n!$ permutation matrices of size n [6].

Left multiplication by a permutation matrix rearranges the corresponding rows. This is then called a row permutation matrix [6]. Here the application is shown on a vector and a matrix:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_2 \\ x_3 \\ x_1 \end{pmatrix} \tag{8}$$

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & a & a \\ b & b & b \\ c & c & c \end{pmatrix} = \begin{pmatrix} b & b & b \\ c & c & c \\ a & a & a \end{pmatrix} \tag{9}$$

Right multiplication by a permutation matrix rearranges the corresponding columns. This is thus called a column permutation matrix [6]. Here it is shown on a matrix:

$$\begin{pmatrix} a & b & c \\ a & b & c \\ a & b & c \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} c & a & b \\ c & a & b \\ c & a & b \end{pmatrix} \tag{10}$$

By multiplying with the row permutation on the left side and the column permutation on the right side, we obtain a full rearrangement of our matrix. This is useful for sorting matrices and will be used in our implementation.

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} = \begin{pmatrix} f & d & e \\ i & g & h \\ c & a & b \end{pmatrix}. \tag{11}$$

4 The SPAI Algorithm

In 1995, Grote and Huckle published *Parallel preconditioning with Sparse Approximate Inverses* [1], which presented an algorithm designed to find the approximate inverse of a sparse matrix by minimising the Frobenius norm. This section will go trough the SPAI algorithm and explain the algorithm in depth.

4.1 Frobenius norm minimisation

We can compute a sparse approximate inverse $M \approx A^{-1}$ via Frobenius norm minimisation. The main idea is to, column for column, minimise the norm and thus obtain a closer approximation of the inverse.

The minimisation

$$\min_M \|AM - I\|_F^2 \tag{12}$$

with $A \in \mathbb{R}^{n \times n}$ and the unity matrix I with the same dimensions is the Frobenius norm minimisation. The smaller the norm is, the closer M will be to the exact inverse of A.

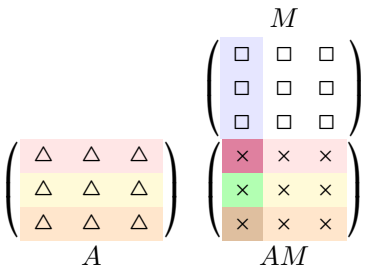

Figure 3: Intuitive example of the matrix multiplication of $AM - I$. Here it is shown clearly how column 0 in the product AM is dependent on the entries of the blue column in M and the both the red, yellow and orange rows of A.

$$\underset{AM}{\begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix}} - \underset{I}{\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}} \approx \underset{AM - I}{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}}$$

Figure 4: Example of how the identity matrix subtracted from the product AM will be approximately 0 if the SPAI algorithm has successfully converged

We can split the minimisation up into a sum of Euclidean norms, like this

$$\min_M \|AM - I\|_F^2 = \sum_{k=1}^{n} \|(AM - I)e_k\|_2^2 \tag{13}$$

$$= \sum_{k=1}^{n} \|Am_k - e_k\|_2^2 \tag{14}$$

where m_k and e_k denotes the k'th column of respectively M and I. Since each summand in (14) represents a least squares problem for every column m_k of M, we can solve them independently from each other. This property gives the algorithm a great advantage by being inherently parallel.

4.2 Computing the sparsity structure $\mathcal{J}$ and $\mathcal{I}$

We want to find a set $\mathcal{J}$ of indices, that represents the current entries we are allowed to update in M. This is defined by the initial sparsity pattern chosen for M. Computing an efficient sparsity pattern, will help us avoid entries, which will only contribute little to M approaching an approximation of A^{-1}.
Let $\mathcal{J}$ be the set of indices j, where m_k has nonzero entries and denote the length of this set by n_2.

$$\mathcal{J} = \{j : m_k(j) \neq 0\} \tag{15}$$
$$n_2 = |\mathcal{J}| \tag{16}$$

We denote the reduced vector $m_k(\mathcal{J})$ by $\hat{m}_k$. Now, because of the sparsity, the reduced system $A(.,\mathcal{J})$ will properly have a lot of zero rows, which is irrelevant for the solution. We eliminate these by letting $\mathcal{I}$, be the set if indices i where $A(i,\mathcal{J})$ is not identically zero

$$\mathcal{I} = \left\{ i : \sum_{j \in \mathcal{J}} |a_{ij}| \neq 0 \right\} \tag{17}$$
$$n_1 = |\mathcal{I}| \tag{18}$$

The resulting submatrix $A(\mathcal{I},\mathcal{J})$, will be denoted by $\hat{A}$.

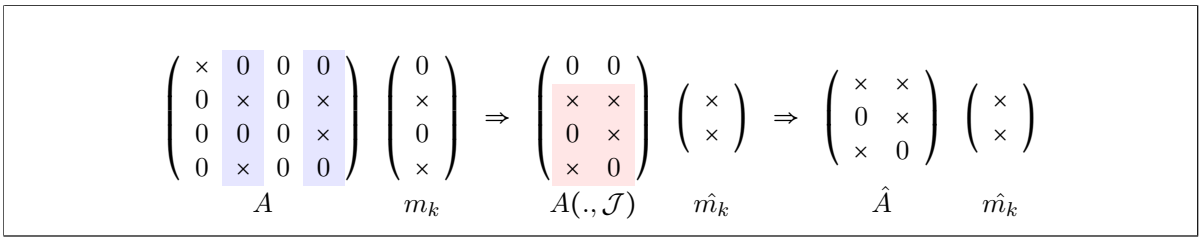

Figure 5: $\mathcal{J}$ is the indices of the blue columns and $\mathcal{I}$ is the indices of the red rows. $\mathcal{J}$ = 1,3 and $\mathcal{I}$ = 1,2,3

We can then reduce the minimisation problem (14) to

$$\min_{\hat{m}_k} ||\hat{A}\hat{m}_k - \hat{e}_k||_2 \tag{19}$$

with the definitions and sizes

$$\hat{A}_k := A(\mathcal{I},\mathcal{J}) \in \mathbb{R}^{n_1 \times n_2}, \qquad \hat{m}_k := m_k(\mathcal{J}) \in \mathbb{R}^{n_2}, \qquad \text{and } \hat{e}_k := e_k(\mathcal{I}) \in \mathbb{R}^{n_1}. \tag{20}$$

The dimensions of this least squares problem is very small, since A and M er both very sparse matrices.

4.3 Applying the Householder QR decomposition

For a the nonsingular matrix A, the submatrix $\hat{A}$ must have full rank [6], and thus we can apply the QR decomposition

$$\hat{A} = Q \begin{pmatrix} R \\ 0 \end{pmatrix}, \tag{21}$$

where

$$Q \in \mathbb{R}^{n_1 \times n_2} \text{ and } R \in \mathbb{R}^{n_2 \times n_2}. \tag{22}$$

We will base our solutions of the minimisation problems on the QR decomposition via Householder reflections, which is an $O(n^3)$ algorithm. See the section concerning QR decomposition with Householder reflections under theoretical prerequisites for further explaining of QR decomposition with Householder reflections.

4.4 Computing the nonzero entries of $\hat{m}_k$

We compute the nonzero entries of $\hat{m}_k$ by

$$\hat{m}_k = R^{-1}\hat{c}, \tag{23}$$

where

$$\hat{c} = Q^T\hat{e}_k \tag{24}$$

We solve this for each $k = 1, ..., n$ and set $m_k(\mathcal{J}) = \hat{m}_k$. This way we have updated the entries of M to be closer to an approximate inverse, which minimises the frobenius norm $\|AM - I\|_F$.

4.5 Improving upon M by updating the sparsity structure

Now we have updated M once, but the goal is to keep improving upon M to obtain an even more precise inverse. We keep iterating and thus improving M until the tolerance is met. To do so, we will augment the sparsity structure and reduce the current error. We will explain this by showing how to update a single column m_k.

We calculate the residual vector r of m_k.

$$r = Am_k - e_k \tag{25}$$

If the vector r only contains zeros, then m_k is already the k'th column of A^{-1} and thus cannot be improved upon.

For the sake of explanation, we assume that $r \neq 0$ and therefore we must augment the set of indices $\mathcal{J}$ to reduce the norm of the residual r. We want to reduce the norm, because the smaller the norm, the closer the approximation.

We denote by $\mathcal{L}$ the set of indices, for which r has nonzero entries. Since A and m_k are sparse, most entries of r will be zero.

$$\mathcal{L} = \{l : r(l) \neq 0\} \cup \{k\} \tag{26}$$

For every $l \in \mathcal{L}$ we can define a set $\mathcal{N}_l$, which consists of the nonzero indices of the l'th row in A

$$\mathcal{N}_l = \{j : a_{lj} \neq 0\} \tag{27}$$

The column indices are the potential new candidates to be added to $\mathcal{J}$, since the other would vanish by multiplying with zero, when computing the matrix-vector product. We will denote these new candidates as

$$\tilde{\mathcal{J}} = \bigcup_{l \in \mathcal{L}} \mathcal{N}_l \tag{28}$$

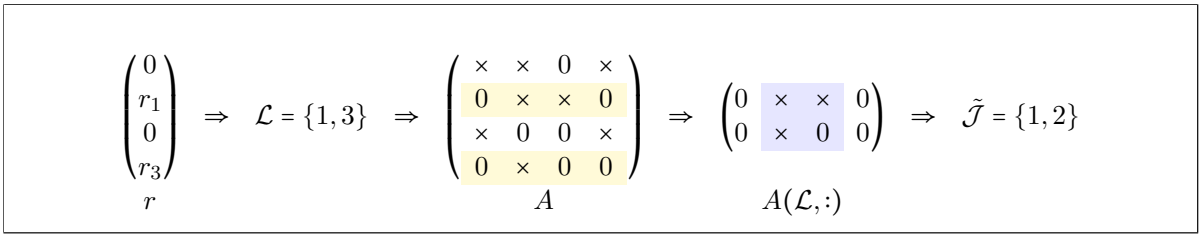

Figure 6: Example of how to find the set $\tilde{\mathcal{J}}$ from the set $\mathcal{L}$

Now we will select the j indices, that will lead to the most profitable reduction in $\|r\|_2$, by solving for each $j \in \mathcal{J}$ the minimisation problem

$$\min_{\mu_j} \|r + \mu_j A e_j\|_2 \tag{29}$$

The solution to this minimisation problem is

$$\mu_j = -\frac{r^T A e_j}{\|Ae_j\|_2^2} \tag{30}$$

In order to see which μ_j would result in the smallest norm of the residual, we calculate the new residuals we would achieve if we added the candidate j to $\mathcal{J}$. So for the new residual $r_{new} = r + \mu_j A e_j$ we compute the 2-norm ρ_j for each j.

$$\rho_j^2 = \|r_{new}\|_2^2 - \frac{(r^T A e_j)^2}{\|Ae_j\|_2^2} \tag{31}$$

When implementing the SPAI algorithm we have created a parameter for the limit of new indices to be added to the current $\mathcal{J}$ in each update step. The algorithm will take this number of indices with the smallest values of ρ_j and union with $\mathcal{J}$. By choosing a large s we ensure fewer iterations until the algorithm converges, but choosing a too large s will result in larger subproblems, that are computationally inefficient. It follows, that there must be at least one index $j \in \tilde{\mathcal{J}}$ that will result in a smaller residual or else we would have reached the tolerance and found the inverse.

Now $\mathcal{J} \cup \tilde{\mathcal{J}}$ contains the column indices of all the nonzero entries of $A(\mathcal{L}, .)$ and $\mathcal{J} \cap \tilde{\mathcal{J}} = \varnothing$. We will use this unionised set to solve the least squares problem again and hence obtain a better approximation of m_k.

With our new column indices $\tilde{\mathcal{J}}$ we can compute the corresponding row indices $\tilde{\mathcal{I}}$. $\tilde{\mathcal{I}}$ is the indices of the rows, which corresponds to the nonzero rows of $A(., \tilde{\mathcal{J}})$ not contained in $\mathcal{I}$ yet. We denote by $\tilde{n_1}$ and $\tilde{n_1}$ the length of the sets $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{J}}$, respectively. We then have to solve the LS problem for the larger submatrix

$$\tilde{A} = A(\mathcal{I} \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) \tag{32}$$

of size $n_1 + \tilde{n_1} \times n_2 + \tilde{n_2}$. To solve this, we will use the QR decomposition of $\hat{A}$, but there is no need to compute the full QR decomposition again. This will be discussed in the next section concerning the update of the QR decomposition.

As long as the norm of the residual is larger than the tolerance provided by the user or the maximal amount of fill-ins has been reached, the update step is repeated until one of the criteria are met. The algorithm will have computed the k'th column of A^{-1} or an approximation as close as the parameters and input would let us. In pseudocode section we present the full SPAI algorithm as pseudocode.

4.6 Updating the QR decomposition

An expensive step in the algorithm is computing the QR decomposition for each iteration of the while-loop, since augmenting the sparsity structure will lead to expanding least squares problems. So instead of performing a new decomposition, it is possible to update the one we already computed. We want to use the available QR decomposition for $\hat{A}$ to update the QR decomposition for $A(\mathcal{I} \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}})$.

First, we compute $\tilde{A}$ with rows from the union of $\mathcal{I}$ and $\tilde{\mathcal{I}}$ and the columns from the union of $\mathcal{J}$ and $\tilde{\mathcal{J}}$.

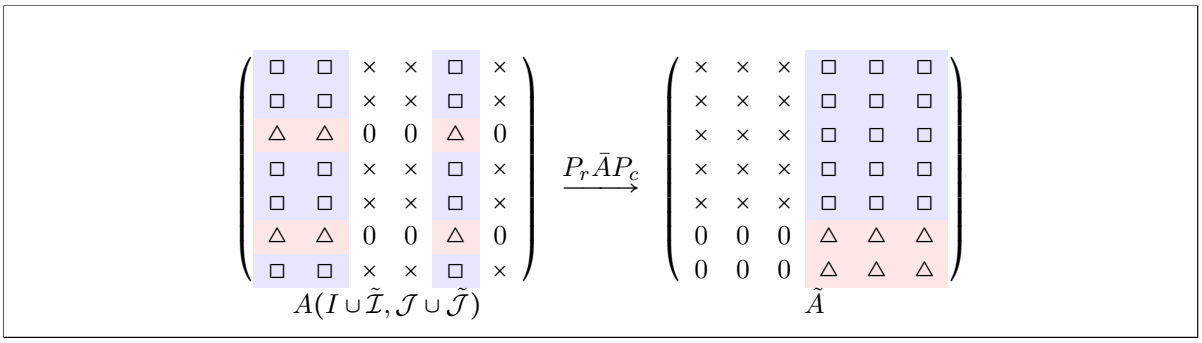

Figure 7: With row and column permutations P_r and P_c, the resulting matrix $\tilde{A}$ will contain a zero block, $\hat{A}$ that is denoted by the ×'s, $A(\mathcal{I}, \tilde{\mathcal{J}})$ that is denoted by □'s and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ that is denoted by △'s. $\tilde{\mathcal{J}}$ is presented with the blue columns and $\tilde{\mathcal{I}}$ is presented the red rows.

$$\tilde{A} = A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) = \begin{pmatrix} \hat{A} & A(\mathcal{I}, \tilde{\mathcal{J}}) \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix} \tag{33}$$

Our $\tilde{A}$ can be rewritten as

$$\tilde{A} = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \begin{pmatrix} R & \breve{A} \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix} \tag{34}$$

where we denote $\breve{A} = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$. This $\breve{A}$ can be split in $Q_1^T A(\mathcal{I}, \tilde{\mathcal{J}})$, which has size $n_2 \times n_2$ and $Q_2^T A(\mathcal{I}, \tilde{\mathcal{J}})$, which has size $\tilde{n_1} + n_1 - n_2 \times n_2$. Simultaneously, R is being split from it's zeros in the bottom. This is how we obtain B_1 and B_2.

$$\tilde{A} = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \left(\begin{array}{c|c} R & \breve{A}(0:n_2, 0:n_2) \\ \hline 0 & \breve{A}(n_2+1:n_1, 0:\tilde{n_2}) \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{array}\right) = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \left(\begin{array}{c|c} R & B_1 \\ \hline 0 & B_2 \end{array}\right) \tag{35}$$

B_1 is an $n_2 \times n_2$ matrix and B_2 is an $\tilde{n_1} + n_1 - n_2 \times \tilde{n_2}$ matrix. As it can be seen above, we only need to compute the QR decomposition of B_2. We let

$$B_2 = Q_B \begin{pmatrix} R_B \\ 0 \end{pmatrix}, \tag{36}$$

Q_B has dimensions $\tilde{n_1} + n_1 - n_2 \times \tilde{n_1} + n_1 - n_2$ and R_B has dimensions $\tilde{n_1} + n_1 - n_2 \times \tilde{n_2}$. Then we obtain

$$\tilde{A} = \begin{pmatrix} Q & \\ & \mathcal{I}_{\tilde{n_1}} \end{pmatrix} \begin{pmatrix} I_{\tilde{n_1}} & \\ & Q_B \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix} \tag{37}$$

from which we can compute Q_B and R_B.

We solve the augmented least squares problem

$$\min_{\mathcal{J} \cup \tilde{\mathcal{J}}} \|\tilde{A}\tilde{M} - \tilde{e}\|_2, \tag{38}$$

where $\tilde{e} = P_r e(\mathcal{I} \cup \tilde{\mathcal{I}})$ and $\tilde{M} = P_c^T M(\mathcal{J} \cup \tilde{\mathcal{J}})$. Finally, we recover the solution in the correct order with a final column permutation $M(\mathcal{J} \cup \tilde{\mathcal{J}}) = P_c \tilde{M}$.

By performing the QR decomposition on the smaller matrix B_2, instead of computing the full QR decomposition of $\tilde{A}$, we reduce the computational cost.

4.7 Pseudocode

Inspired by the SPAI algorithm by Grote and Huckle [1], we have written our own pseudocode, which explains some of the steps of the algorithm more explicitly.

We have based our pseudocode for the substeps of the QR update on algorithm 5 from Sedlacek's dissertation *Sparse Approximate Inverses for Preconditioning, Smoothing, and Regularization* [3].

Algorithm 2: SPAI Algorithm

Input : A: a CSC matrix, tolerance: tolerance , maxIteration: constraint for the maximal number of iterations, s: number of most profitable indices

Output: M: the inverse of A

for *every column* $\hat{m}_k$ *of* M **do**

1) Find initial sparsity $\mathcal{J}$ of $\hat{m}_k$

2) Compute the row indices $\mathcal{I}$ of the corresponding nonzero entries of $A(\mathcal{I}, \mathcal{J})$

3) Create $\hat{A} = A(\mathcal{I}, \mathcal{J})$

4) Do QR decomposition of $\hat{A}$

5) Compute the solution m_k for the least squares problem

5.1) Compute $\hat{c} = Q^T \hat{e}_k$

5.2) Compute R_{-1}

5.3) Compute $\hat{m}_k = R^{-1}\hat{c}$

6) Compute residual

while *residual* > *tolerance* **do**

7) Let $\mathcal{L}$ be the set of indices, where $r(l) \neq 0$

8) Set $\tilde{\mathcal{J}}$ to all new column indices of A that appear in all $\mathcal{L}$ rows, but not in $\mathcal{J}$ yet

9) For each j in $\tilde{\mathcal{J}}$ compute: $\rho_j^2 = \|r_{new}\|_2^2 - \frac{(r^T A e_j)^2}{\|Ae_j\|_2^2}$

10) Find the indices $\tilde{\mathcal{J}}$ corresponding to the smallest s elements of ρ^2

11) Determine the new indices $\tilde{\mathcal{I}}$

12) Make $I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} \cup \tilde{\mathcal{J}}$

13) Update the QR decomposition

13.1) Create $A(I, \tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$

13.2) Compute $\breve{A} = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$

13.3) Compute $B_1 = \breve{A}(0 : n_2, 0 : \tilde{n_2})$

13.4) Compute $B_2 = \begin{pmatrix} \breve{A}(n_2 + 1 : n_1, 0 : \tilde{n_2}) \\ A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix}$

13.5) Do QR decomposition of B_2

13.6) Compute Q and R from

$$A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \begin{pmatrix} I_{n_2} & \\ & Q_B \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix}$$

13.7) Solve the augmented LS problem for $\hat{m}_k$

14) Compute new residual

15) Set $\mathcal{I} = I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} = \mathcal{J} \cup \tilde{\mathcal{J}}$ and $A' = A(\mathcal{I}, \mathcal{J})$

end while

16) Set $m_k(\mathcal{J}) = \hat{m}_k$

end for

4.8 Numerical example

We have made a numerical example that we have traced by hand in order to check the algorithmic accuracy. We have traced the algorithm on the nonsingular 4×4 matrix

$$A = \begin{pmatrix} 10 & 10 & 0 & 14 \\ 0 & 10 & 2 & 0 \\ 13 & 0 & 0 & 1 \\ 0 & 5 & 0 & 0 \end{pmatrix} \tag{39}$$

and used the parameters: tolerance = 0.01, maxIterations = 1 and s = 1. We will show the tracing by hand for column 0 for one iteration. First we define

$$M = I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{40}$$

For column 0, we find $\mathcal{J}$ and $\mathcal{I}$ and with $\mathcal{J}$ and $\mathcal{I}$ we compute $\hat{A} = A(\mathcal{I}, \mathcal{J})$. The blue columns denote the columns with indices in $\mathcal{J}$ and the red rows denote the rows with indices in $\mathcal{I}$.

$$\mathcal{J} = \{0\}, \qquad n_2 = 1 \tag{41}$$

$$\mathcal{I} = \{0, 2\}, \quad n_1 = 2 \tag{42}$$

$$\underset{A}{\begin{pmatrix} 10 & 10 & 0 & 14 \\ 0 & 10 & 2 & 0 \\ 13 & 0 & 0 & 1 \\ 0 & 5 & 0 & 0 \end{pmatrix}} \Rightarrow \underset{A(.,\mathcal{J})}{\begin{pmatrix} 10 \\ 0 \\ 13 \\ 0 \end{pmatrix}} \Rightarrow \underset{\hat{A}}{\begin{pmatrix} 10 \\ 13 \end{pmatrix}} \tag{43}$$

We then compute the QR decomposition of $\hat{A}$.

$$Q = \begin{pmatrix} -0.609711 & -0.792624 \\ -0.792624 & 0.609711 \end{pmatrix}, \quad R = \begin{pmatrix} -16.401220 \\ 0 \end{pmatrix} \tag{44}$$

We then find the solution $\hat{m_k}$ to the least squares problem

$$\hat{m_k} = R^{-1}\hat{c_k} = \begin{pmatrix} -0.060971 \end{pmatrix} \begin{pmatrix} 0.609711 \end{pmatrix} = \begin{pmatrix} 0.037175 \end{pmatrix} \tag{45}$$

Then we compute the residual and the norm of the residual

$$r = A(.,\mathcal{J})\hat{m_k} - e_k = \begin{pmatrix} 10 \\ 0 \\ 13 \\ 0 \end{pmatrix} \begin{pmatrix} 0.037175 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} -0.62853 \\ 0 \\ 0.483275 \\ 0 \end{pmatrix} \tag{46}$$

$$\|r\| = 0.792624 \tag{47}$$

Since the norm of the residual $0.792624 > 0.01$, we go into the while-loop. Let $\mathcal{L}$ be the set of indices, where $r(l) \neq 0$, so since $r(0) = -0.62853$ and $r(2) = 0.483275$

$$\mathcal{L} = \{0, 2\} \tag{48}$$

Then we set $\tilde{\mathcal{J}}$ to all new column indices of A that appear in all $\mathcal{L}$ rows, but not in $\mathcal{J}$ yet. The yellow rows denoted the $\mathcal{L}$ rows and the blue columns denote the nonzero $\tilde{\mathcal{J}}$ columns.

$$\underset{A}{\begin{pmatrix} 10 & 10 & 0 & 14 \\ 0 & 10 & 2 & 0 \\ 13 & 0 & 0 & 1 \\ 0 & 5 & 0 & 0 \end{pmatrix}} \Rightarrow \underset{A(\mathcal{L},:)}{\begin{pmatrix} 10 & 10 & 0 & 14 \\ 13 & 0 & 0 & 1 \end{pmatrix}} \tag{49}$$

$$\tilde{\mathcal{J}} = \{0,1,3\} - \{0\} = \{1,3\} \tag{50}$$

For each j in $\tilde{\mathcal{J}}$ we solve the minimisation problem by computing ρ_j^2.

$$\rho_1^2 = \|r_{new}\|_2^2 - \frac{(r^T A e_1)^2}{\|Ae_1\|_2^2} \tag{51}$$

$$= 0.792624^2 - \frac{\left(\begin{pmatrix}-0.62853 & 0 & 0.483275 & 0\end{pmatrix}\begin{pmatrix}10\\10\\0\\5\end{pmatrix}\right)^2}{15^2} \tag{52}$$

$$= 0.792624^2 - \frac{-6.2853^2}{15^2} \tag{53}$$

$$= 0.452830 \tag{54}$$

$$\rho_3^2 = \|r_{new}\|_2^2 - \frac{(r^T A e_3)^2}{\|Ae_3\|_2^2} \tag{55}$$

$$= 0.792624^2 - \frac{\left(\begin{pmatrix}-0.62853 & 0 & 0.483275 & 0\end{pmatrix}\begin{pmatrix}14\\0\\1\\0\end{pmatrix}\right)^2}{14.04^2} \tag{56}$$

$$= 0.792624^2 - \frac{-8.312267^2}{14.04^2} \tag{57}$$

$$= 0.277523 \tag{58}$$

Then we find the indices $\tilde{\mathcal{J}}$ corresponding to the smallest s = 1 element of ρ^2. Since $0.277523 < 0.452830$ we choose the index 3. We compute the union of $\mathcal{J}$ and $\tilde{\mathcal{J}}$ in order to find $\tilde{\mathcal{I}}$. $\tilde{\mathcal{I}}$ is the indices of the rows, which corresponds to the nonzero rows of $A(.,\mathcal{J}\cup\tilde{\mathcal{J}})$ not contained in $\mathcal{I}$ yet.

$$\begin{array}{ll}\tilde{\mathcal{J}} = \{3\}, & \tilde{n_2} = 1\\ \mathcal{J}\cup\tilde{\mathcal{J}} = \{0,3\}, & n_2\text{union} = 2\\ \tilde{\mathcal{I}} = \{0,2\} - \{0,2\}, = \{\} & \tilde{n_1} = 0\\ \mathcal{I}\cup\tilde{\mathcal{I}} = \{0,2\}, & n_1\text{union} = 2\end{array} \tag{59}$$

The blue columns denote the columns with indices in $\mathcal{J}\cup\tilde{\mathcal{J}}$ and the red rows denote the rows with indices in $\tilde{\mathcal{I}}$ (which there are none of).

$$\underset{A}{\begin{pmatrix}10 & 10 & 0 & 14\\ 0 & 10 & 2 & 0\\ 13 & 0 & 0 & 1\\ 0 & 5 & 0 & 0\end{pmatrix}} \Rightarrow \underset{A(.,\mathcal{J}\cup\tilde{\mathcal{J}})}{\begin{pmatrix}10 & 14\\ 0 & 0\\ 13 & 1\\ 0 & 0\end{pmatrix}} \tag{60}$$

Now it is time to update the QR decomposition. We create $A(I,\tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}},\tilde{\mathcal{J}})$. The blue columns are $\tilde{\mathcal{J}}$ the red rows are respectivelly $\mathcal{I}$ and $\tilde{\mathcal{I}}$. The purple entries are the overlap.

$$\underset{A}{\begin{pmatrix}10 & 10 & 0 & 14\\ 0 & 10 & 2 & 0\\ 13 & 0 & 0 & 1\\ 0 & 5 & 0 & 0\end{pmatrix}} \Rightarrow \underset{A(\mathcal{I},\tilde{\mathcal{J}})}{\begin{pmatrix}14\\ 1\end{pmatrix}} \tag{61}$$

$$\underset{A}{\begin{pmatrix} 10 & 10 & 0 & 14 \\ 0 & 10 & 2 & 0 \\ 13 & 0 & 0 & 1 \\ 0 & 5 & 0 & 0 \end{pmatrix}} \Rightarrow \underset{A(\tilde{\mathcal{I}},\tilde{\mathcal{J}})}{()} \tag{62}$$

We compute

$$\breve{A} = Q^T A(\mathcal{I},\tilde{\mathcal{J}}) = \begin{pmatrix} -0.609711 & -0.792624 \\ -0.792624 & 0.609711 \end{pmatrix}\begin{pmatrix} 14 \\ 1 \end{pmatrix} = \begin{pmatrix} -9.328573 \\ -10.487024 \end{pmatrix} \tag{63}$$

And then we create $B_1 = \breve{A}(0:1, 0:1)$.

$$B_1 = \begin{pmatrix} -9.328573 \end{pmatrix} \tag{64}$$

We create $B_2 = \breve{A}(1+1:2, 0:1)$ above $A(\tilde{\mathcal{I}},\tilde{\mathcal{J}})$. $A(\tilde{\mathcal{I}},\tilde{\mathcal{J}})$ is empty in this case.

$$B_2 = \begin{pmatrix} -10.487024 \end{pmatrix} \tag{65}$$

Then we do the QR decomposition of B_2

$$Q_B = \begin{pmatrix} 1 \end{pmatrix}, \quad R_B = \begin{pmatrix} -10.487024 \end{pmatrix} \tag{66}$$

We make the new Q and R

$$Q = \begin{pmatrix} Q & \\ & I_2 \end{pmatrix}\begin{pmatrix} I_1 & \\ & Q_B \end{pmatrix} = \begin{pmatrix} -0.609711 & -0.792624 \\ -0.792624 & 0.609711 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} -0.609711 & -0.792624 \\ -0.792624 & 0.609711 \end{pmatrix} \tag{67}$$

$$R = \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} -16.401220 & -9.328573 \\ 0 & -10.487024 \end{pmatrix} \tag{68}$$

And then we solve the augmented least squares problem for $\hat{m_k}$

$$\hat{m_k} = R^{-1}\hat{c} = \begin{pmatrix} -0.060971 & 0.054236 \\ 0 & -0.095356 \end{pmatrix}\begin{pmatrix} -0.0609711 \\ -0.792624 \end{pmatrix} = \begin{pmatrix} -0.005814 \\ 0.075581 \end{pmatrix} \tag{69}$$

We compute the new residual

$$r = A(.,\mathcal{J}\cup\tilde{\mathcal{J}})\hat{m_k} - e_k = \begin{pmatrix} 10 & 14 \\ 0 & 0 \\ 13 & 1 \\ 0 & 0 \end{pmatrix}\begin{pmatrix} -0.005814 \\ 0.075581 \end{pmatrix} - \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \tag{70}$$

$$\|r\| = 0 \tag{71}$$

With the norm of the residual being 0 and thus smaller than the tolerance, there is no need to perform more iterations. We have reached the optimal solution for column 0 and $M(:,0)$ is approximately $A^{-1}(:,0)$.

5 Implementation

The previous section was devoted to the theoretical development and presentation of the SPAI algorithm. In this section we will focus on the implementation of the SPAI preconditioner in both a sequential and parallel environment. We will show code snippets of relevant parts of the code. We are aware that some of the code snippets are a bit long, but from our thousands of lines of code, we have deemed these lines the most important.

5.1 Sequential implementation in Python

We have implemented the SPAI algorithm sequentially in Python. The implementation works as a prototype for our final implementation in CUDA. We implemented the prototype as the first thing, as a concrete way of understanding the algorithm in depth.

5.1.1 Structure

The structure of the implementation follows the pseudocode rigorously. We defined a function `SPAI(A, tol, max_iter, s)`, which takes the inputs: A CSC matrix, a tolerance, a constraint for the maximal number of iterations and the number of most profitable indices. The inputs can be defined by the user, when calling the function.

5.1.2 Choices and shortcomings

Checking if A is nonsingular
Before we begin the steps of the algorithm, we have to check if A is nonsingular. If A is singular, it cannot be inverted and there is no point in finding the approximate inverse. We compute the determinant and check if it is nonzero, meaning that A is nonsingular and thus invertible. A turned out primarily to be singular if the matrix was both very small and very sparse, making the chance of zero rows or columns large.

M is set to an identity CSC matrix
We chose that the initial sparsity pattern for M should be the identity matrix, inspired by Grote and Huckle's implementation [1]. We used the `scipy.sparse` library to create an identity matrix in the CSC format [12].

Dense submatrices for the QR decomposition
The input of our function is a sparse matrix A and we perform most of the steps of our algorithm on A in the sparse format. However, QR decomposition is not implemented in Python for sparse matrices. We densify the subproblems used for the QR decomposition. The first QR decomposition is performed for each column of M on the submatrix $\hat{A}$. Therefore we densify $\hat{A}$.

```
# 13) Do QR decomposition of AHat
Q, R = np.linalg.qr(AHat.todense(), mode="complete")
```

Listing 1: QR composition of $\hat{A}$

The second QR decomposition is performed inside the while-loop on the matrix B_2, that we create by stacking the lower part of $\breve{A}$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$. We densify the submatrix $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ in order to use it in B_2.

```
# 13.4) Compute B2 = ABreve(n2 + 1 : n1, 0 : n2Tilde) above AITildeJTilde
B_2 = np.vstack((ABreve[n2:n1,:], AITildeJTilde.todense())

# 13.5) Do QR decomposition of B2
Q_B, R_B = np.linalg.qr(B_2, mode="complete")
```

Listing 2: Computing B_2 and performing the QR decomposition of B_2

Permutation matrices
When we want to sort matrices or vectors, we use permutation matrices. We create the row permutation matrix from the unionised set of $\mathcal{I}$ and $\tilde{\mathcal{I}}$ and the column permutation from the unionised set of $\mathcal{J}$ and $\tilde{\mathcal{J}}$. We use the permutation matrices to sort Q and R, so they are in the correct order for the next iteration of the for-loop.

```
Q = Pr * Q
R_1 = R_1 * Pc
```

Listing 3: Permuting Q and R

Singular matrix R
In order to solve the least squares problem for m_k, we need to compute the inverse of R. This cannot be done if R is singular. A singular matrix has a determinant equal to zero and thus is not invertible. When running the SPAI function on a sparse matrix A smaller than 40×40, R turns out to be singular sometimes. With R not being invertible, we cannot solve the LS problem for that column and the approximation of the inverse will not be close to the actual inverse of that column.

```
# 5) Compute the solution m_k for the least sqaures problem
# 5.1) Compute cHat = Q^T * eHat_k
e_k = np.matrix([0]*M.shape[1]).T
e_k[k] = 1

eHat_k = e_k[I]
cHat_k = Q.T * eHat_k

# 5.2) Compute the inverse of R
invR_1 = np.linalg.inv(R_1)

# 5.3) Compute mHat_k = R^-1 * cHat
mHat_k = invR_1 * cHat_k[0:n2,:]

m_k[J] = mHat_k
```

Listing 4: Solving the LS problem

However, this turned out to be a rare issue after checking if A was nonsingular and thus only running the algorithm on nonsingular A matrices.

Smallest indices
Right now the part of our implementation, which should find the s indices with the smallest ρ^2 values and only add those to $\tilde{\mathcal{J}}$, does not work. Our implementation adds all potential candidates to $\tilde{\mathcal{J}}$. We want to ideally only choose the indices, that leads to the most profitable reduction of the residual and that is the indices with the smallest ρ^2 value. So adding all indices, will potentially lead to more computations without much improvement. The missing implementation of the smallest indices will lead to more work, but not worse accuracy.

5.2 Sequential implementation in C

After implementing the prototype in Python, we implemented the SPAI algorithm sequentially in C with the use of cuBLAS library functions. Therefore, all the code is written in `cu.h` CUDA header files. In the following subsections we will present the functions we have created in order to implement the algorithm and discuss our choices and shortcomings.

5.2.1 The sequentialSpai function

We have implemented the algorithm as a function `CSC* sequentialSpai(CSC* A, float tolerance, int maxIteration, int s)` that takes the inputs: A CSC matrix, a tolerance, a constraint for the maximal number of iterations and the number of most profitable indices and returns a CSC matrix.

The structure of the implementation follows the steps from our pseudocode. We have implemented functions that deals with repeated parts of the algorithm, such as computing the QR decomposition and the least squares problem. We have also implemented a function for updating the QR decomposition and thus performing the 7 substeps of that process. All the subfunctions returns 0 if they were successful and 1 if not, making it more manageable when testing. Here is the structure of the main function `sequentialSPAI`, which can be found in the file `sequentialSpai.cu.h`:

Algorithm 3: Structure of sequential CUDA implementation

Input : A: a CSC matrix, tolerance: tolerance , maxIteration: constraint for the maximal number of iterations, s: number of most profitable indices
Output: M: the inverse of A
for *every column* $\hat{m}_k$ *of* M **do**
  1) Find initial sparsity $\mathcal{J}$ of $\hat{m}_k$
  2) Compute the row indices $\mathcal{I}$ of the corresponding nonzero entries of $A(\mathcal{I}, \mathcal{J})$
  3) Create $\hat{A} = A(\mathcal{I}, \mathcal{J})$
  4) Do QR decomposition of $\hat{A}$
  `qrBatched(cHandle, &AHat, n1, n2, &Q, &R)`
  5) Compute the solution m_k for the least squares problem
  `LSProblem(cHandle, A, Q, R, &mHat_k, residual, I, J, n1, n2, k, &residualNorm)`
  6) Compute residual
  while *residual* < *tolerance* **do**
    7) Let $\mathcal{L}$ be the set of indices, where $r(l) \neq 0$
    8) Set $\tilde{\mathcal{J}}$ to all new column indices of A that appear in all $\mathcal{L}$ rows, but not in $\mathcal{J}$ yet
    for *each* j *in* $\tilde{\mathcal{J}}$ **do**
      9) Compute ρ_j^2
    end for
    10) Find the indices $\tilde{\mathcal{J}}$ corresponding to the smallest s elements of ρ^2
    11) Determine the new indices $\tilde{\mathcal{I}}$
    12) Make $I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} \cup \tilde{\mathcal{J}}$
    13) Update the QR decomposition
    `updateQR(cHandle, A, &AHat, &Q, &R, &I, &J, ITilde, JTilde, IUnion, JUnion, n1, n2, n1Tilde, n2Tilde, n1Union, n2Union, &mHat_k, residual, &residualNorm, k)`
    14) Compute the residual
    15) Set $\mathcal{I} = I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} = \mathcal{J} \cup \tilde{\mathcal{J}}$ and $A' = A(\mathcal{I}, \mathcal{J})$
  end while
  16) Set $m_k(\mathcal{J}) = \hat{m}_k$
  `updateKthColumnCSC(M, mHat_k, k, J, n2)`
end for

Before the for-loop begins, we perform three important steps: The first step of the sequential SPAI function is, similarly to python implementation, checking if A is nonsingular. We do so, by performing a LU factorisation with a CuSolver library function and checking if it succeeds. We have done so with the function `checkSingularity` in the file `singular.cu.h`, which returns 1 if A is singular and 0 otherwise. This is very time consuming for large matrices, so when we test the run time of the implementations, we exclude this part as it is not a part of the algorithm itself. For future optimisations, we would implement a cheaper way of checking if A is singular.

We then initialise cuBLAS, which is calling the cuBLAS library function later in the code. We use the same cuBLAS handle for all computations and destroy it in the very end.

Then we initialise M, which is our output matrix and in the end approximate inverse of A. We

have chosen to set M to the identity matrix. One could choose the initial sparsity of M many different ways, but we have been inspired by the Grote and Huckle [1] who sets M to the identity.

Then we initialise the for-loop and for each iteration of it, we perform the steps described in the structure of the function. Right before the while-loop, we compute the residual, which is used as one of the stopping condition for the while-loop. When the while-loop terminates, either by the tolerance being met or the max number iterations being reached, we update the k'th column in M with $\hat{m}_k$. We do so with the function `updateKthColumnCSC`. Then we move on to the next column of M.

5.2.2 The CSC format

The purpose of the Sparse Approximate Inverse algorithm is to find the approximate inverse of a sparse matrix. Thus we had to create a format for sparse matrices. We implemented the CSC storage format, which was explained in the background section. We chose to make the storage format column-major, since we operate on columns in M, meaning we get coalesced memory access.

In the file `csc.cu.h` we made a struct representing a sparse matrix called CSC, which will be used throughout the rest of the implementation.

```
typedef struct CSC {
    int m;
    int n;
    int countNonZero;
    int* offset;
    float* flatData;
    int* flatRowIndex;
} CSC;
```

We made functions for:

- Creating a CSC from a dense matrix
- Creating a CSC identity matrix
- Creating a CSC consisting of random floats
- Updating the k'th column of a CSC
- Making a CSC into a dense matrix
- Multiplying two CSC's
- Printing a CSC
- Freeing a CSC

These functions will be widely used in the implementation and especially in the testing of the code.

5.2.3 The qrBatched function: Computing the QR decomposition

An important part of finding the approximate inverse with the SPAI algorithm is performing the QR decomposition. As explained in the theoretical prerequisites for SPAI, we compute the QR decomposition with Householder reflections. We created a function for computing the QR decomposition called `qrBatched`. In the sequential version the batchsize is simply 1. The function is used to initially compute the QR decomposition of $\hat{A}$ and later called inside `updateQR` to compute the QR decomposition of B_2. The example explained here is for the $\hat{A}$ case.
Here is the structure of the function in the file `qrBatched.cu.h`:

Algorithm 4: Structure of qrBatched function

Input : cHandle:the cuBLAS handle, AHat: an array of pointers to batch matrices, n1: the max number of rows of the matrices, n2: the max number of columns of the matrices, Q: an array of pointers to batch Q matrices, R: an array of pointers to batch R matrices

Output: int 0 for success or int 1 for failure

Create input and output arrays

Malloc space and copy data for AHat and tau

Run QR decomposition from cuBLAS

`cublasSgeqrfBatched(cHandle, n1, n2, d_PointerAHat, lda, d_PointerTau, &info, BATCHSIZE);`

Copy AHat and tau back to host

Copy R from AHat

Make Q with Algorithm 1

We use the cuBLAS library function `cublasSgeqrfBatched`[10]. We use the batched cuBLAS function in the sequential version in order to make the transition to the parallel version easier to implement.

Our d_PointerAHat is an array of pointers to matrices with dimensions $m \times n$. TauArray is an array of pointers to vectors of dimension of at least max $(1, \min(m, n))$.

In order to use AHat and tau from the device memory, we create d_AHat and d_tau and allocate space for them. We copy the data for AHat from the host to the device. Since the cuBLAS function takes a array of pointers to AHat, we created a kernel that creates an array of pointers d_PointerAHat, which points to the start of each submatrix in d_AHat.

```
__global__ void AHatDeviceToDevicePointerKernel(float** d_AHat, float*
    h_AHat, int batch, int n1, int n2) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < BATCHSIZE) {
        d_AHat[tid] = &h_AHat[tid * n1 * n2];
    }
}
...
AHatDeviceToDevicePointerKernel <<< 1, BATCHSIZE >>> (d_PointerAHat, d_AHat
    , BATCHSIZE, n1, n2);
```

Listing 5: The device to device pointer kernel for used for $\hat{A}$

The same principle is implemented for tau. After using the cuBLAS function, we copy AHat and tau back to host.

The R is stored in the upper triangular part of AHat and is easily extracted with for-loops. The Q matrix is represented as a product of elementary reflections and is stored in the lower part of AHat. The Q is computed with algorithm 1 [13].

The cuBLAS documentation states: "This function is intended to be used for matrices of small sizes, where the launch overhead is a significant factor" [10]. Even though the input of the algorithm is a potentially very large matrix, because of the sparsity, we will perform the QR decomposition on small subproblems.

5.2.4 The LSproblem function: Solving the least squares problem

Another essential part of the algorithm is the solution to the least squares problem. In the pseudocode we have denoted that step as number 5), with substeps 1-3. We have created a function `LSproblem`

that computed those three steps. The `LSproblem`-function is called both in the initial part of the algorithm and in the while-loop. Here is the structure of the function in the file `LSproblem.cu.h`:

Algorithm 5: Structure of LSProblem function

Input : cHandle: the cuBLAS handle, A: the original CSC matrix, Q: the Q matrix from the QR factorization of AHat, R: the R matrix from the QR factorization, mHat_k: the solution of the least squares problem, residual: the residual vector of the least squares problem, I: indices of the rows of AHat, J: indices of the columns of AHat, n1: the number of rows of AHat, n2: the number of columns of AHat, k: the index of the column of mHat, residualNorm: the norm of the residual vector

Output: int 0 for success or int 1 for failure

5.1) Compute $\hat{c} = Q^T \hat{e}_k$

5.2) Compute R_{-1}

`invBatched(cHandle, &R, n2, &invR)`

5.3) Compute $\hat{m}_k = R^{-1}\hat{c}$

The function mostly consists of matrix-vector multiplication done with for-loops. Here is an example of how we compute $\hat{c} = Q^T \hat{e_k}$.

```
float* cHat = (float*) malloc(n2 * sizeof(float));
for (int j = 0; j < n2; j++) {
    cHat[j] = 0.0;
    for (int i = 0; i < n1; i++) {
        cHat[j] += Q[i * n1 + j] * e_k[i];
    }
}
```

Listing 6: Sequential computation of $\hat{c}$

In order to compute the inverse of R, we have implemented a function `invBatched`, which performs the inversion and returns it on the address of invR.

5.2.5 The invBatched function: Computing the inverse of R

In order to solve the LS problem, we have to compute the inverse of R. Here is the structure of the `invBatched` function:

Algorithm 6: Structure of invBatched function

Input : cHandle:the cuBLAS handle, A: an array of pointers to batch matrices, n: the max number of rows and columns of the matrices, invA: an array of pointers to batch inverse matrices

Output: int 0 for success or int 1 for failure

Create input and output arrays

Malloc space and copy data for A

Malloc space for invA, pivot array and info

Run batched LU factorization from cuBLAS

`cublasSgetrfBatched(cHandle, n, d_PointerA, lda, d_PivotArray, d_info, BATCHSIZE);`

Run batched inversion from cuBLAS

`cublasSgetriBatched(cHandle, n, d_PointerA, lda, d_PivotArray, d_PointerAInv, ldc, d_info, BATCHSIZE)`

Copy result back to host

We use the cuBLAS library function `cublasSgetriBatched` [10] to compute the inverse. A and AInv are arrays of pointers to matrices stored in column-major format with dimensions $n \times n$. Similarly to `qrBatched`, we have to create d_A and d_AInv and malloc space for them in order to use A and AInv from the device memory. Because the batched `cublasSgetriBatched`-function takes A and AInv as arrays of pointers, we transform them from device to device pointers. We have created a helper kernel for this process:

```
__global__ void deviceToDevicePointerKernel(float** d_PointerA, float* d_A,
    int batch, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < BATCHSIZE) {
        d_PointerA[tid] = &d_A[tid * n * n];
    }
}
...
deviceToDevicePointerKernel <<< 1, BATCHSIZE >>> (d_PointerA, d_A,
   BATCHSIZE, n);
```

Listing 7: Kernel for copying the start of a device array to a device pointer array

The cuBLAS inversion function only works on upper triangular matrices. Thus we have to transform the input matrix to upper triangular form by computing the LU factorisation. We do so with the `cublasSgetrfBatched` [10] function from cuBLAS. After doing both the LU factorisation and the inversion, we copy d_AInv back to the host, so we can use it in the rest of the implementation that runs on the CPU. Ideally, we would want to avoid using the LU decomposition, since the matrix is already on upper triangular form. The LU decomposition, however, does changes to the upper triangular matrix, that are necessary for the inversion function to compute the correct inverse.

5.2.6 The updateQR function: Updating the QR decomposition

Updating the QR decomposition is an important step in the algorithm. Instead of computing the full QR decomposition of the enlarged subproblem $A(\mathcal{I} \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}})$ each time, we only compute new Q and R matrices for a certain part and augment it with the old Q and R. We do so in order to save a potential huge amount of computationally effort. The updating of Q and R takes multiple steps and thus we have created a function for doing so. The structure of the function follows step 13.1-13.7 in the pseudocode. Here is the structure of the function in the file `udpateQR.cu.h`:

Algorithm 7: Structure of updateQR function

Input : cHandle: the cuBLAS handle, A: the original CSC matrix, AHat: the submatrix of size n1 x n2, Q: the Q matrix, R: the R matrix, I: the row indices of AHat, J: the column indices of AHat, ITilde: the row indices to potentially add to AHat, JTilde: the column indices to potentially add to AHat, IUnion: the union of I and ITilde, JUnion: the union of J and JTilde, n1: the length of I, n2: the length of J, n1Tilde: the length of ITilde, n2Tilde: the length of JTilde, n1Union: the length of IUnion, n2Union: the length of JUnion, m_kOut: the output of the LS problem, residual: the residual vector, residualNorm: the norm of the residual vector, k: the current iteration

Output: int 0 for success or int 1 for failure

13.1) Create $A(I, \tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$

```
createPermutationMatrices(IUnion, JUnion, n1Union, n2Union, Pr, Pc)
```

13.2) Compute $\breve{A} = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$

13.3) Compute $B_1 = \breve{A}(0 : n_2, 0 : n_2)$

13.4) Compute $B_2 = \breve{A}(n_2 + 1 : n_1, 0 : \tilde{n_2})$ above $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$

13.5) Do QR decomposition of B_2

```
qrBatched(cHandle, &B2, n1Union - n2, n2Tilde, &B2Q, &B2R))
```

13.6) Compute Q and R

13.7) Solve the augmented LS problem for $\hat{m_k}$

```
LSProblem(cHandle, A, unsortedQ, unsortedR, m_kOut, residual, IUnion, JUnion,
 n1Union, n2Union, k, residualNorm)
```

To begin with we create $A(I, \tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$. We use our function `CSCToDense` to create the dense submatrices with the given indices. Then we create the row and column permutation matrices with the function `createPermutationMatrices` from the file `permutation.cu.h`.

We then compute $\breve{A}$ and slice the matrix in order to obtain B_1 and the upper part of B_2. B_2 is created by stacking $\breve{A}(n_2 + 1 : n_1, 0 : \tilde{n_2})$ on top of $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ with for-loops. We then perform the QR decomposition on B_2 using our `qrBatched` function. We compute the new Q and R from

$$A(I \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) = \begin{pmatrix} Q & \\ & I_{\tilde{n_1}} \end{pmatrix} \begin{pmatrix} I_{n_2} & \\ & Q_B \end{pmatrix} \begin{pmatrix} R & B_1 \\ 0 & R_B \\ 0 & 0 \end{pmatrix}$$

The resulting Q and R are unsorted, meaning that the row and columns are not permuted. We solve the LS problem with the unsorted Q and R. From this we obtain the unsorted $\hat{m_k}$, which we then sort by multiplying the column permutation matrix with unsorted $\hat{m_k}$.

Previously we sorted Q and R in each iteration, but that was not necessary, since the solution to the LS problem is still the correct $\hat{m_k}$, just in the wrong order. This means we only have to permute $\hat{m_k}$ and sort $\mathcal{J}$. Instead of having to permute Q, R, $\mathcal{I}$ and $\mathcal{J}$, which is two matrices and two vectors, we only have to permute two vectors, resulting in lower time complexity.

5.3 Parallel implementation in CUDA

The final implementation we implemented was the parallel version of the SPAI algorithm in CUDA. The implementation follows the structure of the pseudocode. The algorithm is inherently parallel because each column of the matrix can be solved as a separate sub problem and put together in the end. The idea is to compute the approximate inverse of each row of A stored in the respective column of M.

We batch each subproblem. The advantage of making functions batched are the full utilisatión

of threads. When parallelising, we use a thread for each entry of a matrix, so when dealing with small matrices, we might only use a small amount of threads. For an example a 2×2 matrix would only use 4 threads and the overhead of running a kernel with 4 threads would be too large compared to the efficiency. Batching means we compute a number of subproblems at the same time. If we have a batchsize of 4, we can compute 4 matrices of e.g. size 2×2, which uses 16 threads instead of the 4 for each matrix. Depending on the hardware a greater or small batchsize can be chosen, so we do not initialise more threads than is physically present in the hardware, or use more memory, than the device has.

5.3.1 Kernels in CUDA

In CUDA it is possible to make a type of function called a kernel, which when called, executes the code in the function, n times in parallel by n different CUDA threads. In CUDA a kernel is defined using declaration specifier `__global__`. CUDA then provides functions to get the ID of the thread, the ID of the block of threads, and the dimensions of the block. This makes us able to distinguish between threads, and thus, it makes us able to create unique index values for arrays. Threads in CUDA are stored in blocks of a given size. If we want to use more threads than one block have, we need to use more blocks. Below is an example of how to define a kernel.

```
__global__ void example(int* input1, int* input2) {
    int tid = blockIdx.x * BlockDim.x + threadIdx.x;
    input1[tid] = input2[tid] + 1;
}
```

Listing 8: An example of how to define a kernel

The kernel is then called later on by giving it the normal function arguments as well as also giving it the number of block it has to use and how big those block are. Another CUDA function returns the blocksize of the hardware used. We can use this to calculate how many blocks to use[9].

```
// int* A of size n, and int* b of size n
// Calculate the number of blocks
int numBlocks = (n + blocksize - 1) / blocksize;

// Run the kernel
example <<<numBlocks, blocksize>>> (A, B)
```

Listing 9: An example of how to run a kernel

Kernels in CUDA runs on graphic processing units (GPUs), which have their own memory. For the device to be able to do any computations, we have to copy the memory of the data we want to work on to the device. CUDA provides functions for this. If we want to to copy an array from host memory (CPU memory) to the device memory (GPU memory), we would do the following:

```
float* d_A;

cudaMalloc((void**) &d_A, n * sizeof(float));
cudaMemcpy(d_A, h_A, n * sizeof(float), cudaMemcpyHostToDevice);
```

Listing 10: Allocating and copying memory to the device

In this example, we assume we have an array in CPU memory called h_A of size n. We then declare a new variable d_A, in which we want to store the data from h_A on device memory. Space is allocated for d_A on the device with the function cudaMalloc, and then the CUDA function cudaMemcpy is used to copy the data from h_A to d_A. If we then run a kernel, that changes the values in d_A and we want to use this data on the host again, we would have to copy it back. We do this the following way:

```
float* h_A = (float*) malloc(n * sizeof(float));

cudaMemcpy(h_A, d_A, n * sizeof(float), cudaMemcpyDeviceToHost);
```

Listing 11: Allocating memory on the host and copying memory from device to host

Here we allocate memory on the host with the C function malloc. If h_A already existed, we would have to free the memory of it first. Then we use cudaMemcpy to copy the memory from the device to the host. The data from d_A is now usable on the host in h_A.

If we for an example want to add a set of batchsize number of matrices with another set of batchsize number of matrices and save it in a third set of batchsize number of matrices, where all the matrices have the size $m \times n$ and where the memory of the matrices are already on the device, we would do the following:

```
// Add batch matrices kernel
__global__ void addBatchMatrices(float** pointerA, float** pointerB, float
    ** pointerC, int m, int n, int batchsize) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < m * n * batchsize) {
        int b = tid / (m * n);
        int i = (tid % (m * n)) / n;
        int j = (tid & (m * n)) % n;

        float* A = pointerA[b];
        float* B = pointerB[b];
        float* C = pointerC[b];

        C[i * n + j] = A[i * n + j] + B[i * n + j];
    }
}

// Run the kernel
int numBlocks = (m * n * batchsize + blocksize - 1) / blocksize;
addBatchMatrices <<<numBlocks, blocksize>>> (A, B, C)
```

Listing 12: An example kernel of how to index

In this example, we start by getting the unique thread ID. With this we then calculate a unique set of index values. b for the index of the matrix in the array of pointers to arrays, i for the row index and j for the column index. We can then copy the pointer to the start of the b'th array, by dereferencing the pointer of the pointer array with b. We can then index in each of the batch matrices as normal.

If we were to write this function sequentially, we would do the following:

```
void* addBatchMatrices(float* A, float* B, float* C, int m, int n, int
    batchsize) {
    for (int b = 0; b < batchsize; b++) {
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                C[b * m * n + i * n + j] = A[b * m * n + i * n + j]
                                         + B[b * m * n + i * n + j];
            }
        }
    }
}
```

Listing 13: The corresponding sequential code

Since we have 3 nested for loops, this would run in $O(batchsize \cdot m \cdot n)$ time, where as the parallel version would run in $O(1)$ time, assuming we have at least $batchsize \cdot m \cdot n$ threads available.

5.3.2 The parallelSpai function and the SPAI kernels

The `parallelSpai` function is the main function of the implementation. It's job is to pass the right variables from the right memory storage to the kernels and the subfunctions.

The first thing we do is initialise the cuBLAS handle to use throughout the rest of the implementation. We then set M to be the identity matrix.

We copy the A and M matrices to the device. For this we made an extra CSC-function in `csc.cu.h`, that uses cudaMalloc and cudaMemcpy to copy from the host to the device. We need all data we are working on to be on the GPU memory. When we, in the first step of the for-loop, allocate space for $\mathcal{I}$ and $\mathcal{J}$ or more precisely the pointer arrays that point to the start of each $\mathcal{I}$ and $\mathcal{J}$ array for each subproblem, use cudaMalloc. The function follows this structure:

Algorithm 8: Structure of parallelSPAI implementation

Input : A: a CSC matrix, tolerance: tolerance , maxIteration: constraint for the maximal number of iterations, s: number of most profitable indices, batchsize: number of matrices to be processed in parallel

Output: M: the inverse of A

for $i <$ *numberOfBatches* **do**

- 1) Find initial sparsity $\mathcal{J}$ of $\hat{m}_k$
- 2) Compute the row indices $\mathcal{I}$ of the corresponding nonzero entries of $A(\mathcal{I}, \mathcal{J})$
- Kernel: `computeIandJ`
- 3) Create $\hat{A} = A(\mathcal{I}, \mathcal{J})$
- Kernel: `CSCToBatchedDenseMatrices`
- 4) Do QR decomposition of $\hat{A}$
- Function: `qrBatched`
- 5) Compute the solution m_k for the least squares problem
- Kernel: `LSProblem`
- **while** *residual* $<$ *tolerance and iterations* $<$ *maxIterations* **do**
  - 7) Let $\mathcal{L}$ be the set of indices, where $r(l) \neq 0$
  - Kernel: `computeLengthOfL`
  - 8) Set $\tilde{\mathcal{J}}$ to all new column indices of A that appear in all $\mathcal{L}$ rows, but not in $\mathcal{J}$ yet
  - Kernel: `computeKeepArray`
  - Kernel: `computeN2Tilde`
  - Kernel: `computeJTilde`
  - 9) Compute ρ_j^2
  - Kernel: `computeRhoSquared`
  - 10) Find the indices $\tilde{\mathcal{J}}$ corresponding to the smallest s elements of ρ^2
  - Kernel: `computeSmallestIndices`
  - 11) Determine the new indices $\tilde{\mathcal{I}}$
  - Kernel: `computeITilde`
  - 12) Make $\mathcal{I} \cup \tilde{\mathcal{I}}$ and $\mathcal{J} \cup \tilde{\mathcal{J}}$
  - 13) Update the QR decomposition
  - Function: `updateQR`
  - 15) Set $\mathcal{I} = \mathcal{I} \cup \tilde{\mathcal{I}}$ and $\mathcal{J} = \mathcal{J} \cup \tilde{\mathcal{J}}$ and $A' = A(\mathcal{I}, \mathcal{J})$
  - Kernel: `copyIandJ`
- **end while**
- 16) Set $m_k(\mathcal{J}) = \hat{m}_k$
- Kernel: `updateBatchColumnsCSC`

end for

The very first step of the pseudocode is the for-loop, but we will not implement it, since we parallelise the algorithm instead. We loop through the batches. We start by identifying how many columns there are in M. `parallelSpai` takes a batchsize, which is the number of columns we want to compute in parallel. Since we only compute the batchsize number of columns in parallel, we compute how many iteration we have to run through, until all the columns have been computed. We do this by dividing the number of columns with the batchsize and round up.

To achieve parallelism, the entire implementation consists of kernels and subfunctions that call other kernels. These kernels and subfunctions are designed to execute in parallel, allowing for efficient utilisation of computational resources and improved performance. Thus we have made a file `SPAIkernels.cu.h` that stores the kernels used in the function `parallelSpai`.

We have also made some helper kernels in `helperKernels.cu.h` that are used multiple times throughout the implementation. We have made a kernel to copy device data to device pointer.

```
__global__ void floatDeviceToDevicePointerKernel(float** d_Pointer, float*
    d_data, int pointerArraySize, int dataSize) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < pointerArraySize) {
        d_Pointer[tid] = &d_data[tid * dataSize];
    }
}
```

Listing 14: Kernel for copying the start of each array in the batch to a pointer array

The kernel takes the arguments; d_Pointer, which is an array of pointers to the start of each matrix in d_data and then d_data, which is an array of batch matrices, padded with zeros, so they have the same size. It also takes the size of d_Pointer and the size of each matrix in d_data. We can then use the d_Pointer array to get the pointer to the start of the the specific matrix in the batch each thread has to work on, as described in section about kernels in CUDA.

The first kernel `computeIAndJ` performs step 1 and 2 of the pseudocode and computes $\mathcal{I}$ and $\mathcal{J}$ as the name implies. This is not the most interesting kernel, so futher explanation of it has been omitted.

The next step is creating $\hat{A}$ with the kernel `CSCToBatchedDenseMatrices`. Looking at the sequential code for this function:

```
float* CSCToDense(CSC* csc, int* I, int* J, int n1, int n2) {
    float* dense = (float*) calloc(n1 * n2, sizeof(float));

    for (int i = 0; i < n1; i++) {
        for(int j = 0; j < n2; j++) {
            for (int l = csc->offset[J[j]]; l < csc->offset[J[j] + 1]; l++)
    {
                if (I[i] == csc->flatRowIndex[l]) {
                    dense[i * n2 + j] = csc->flatData[l];
                }
            }
        }
    }

    return dense;
}
```

Listing 15: Sequential implementation of setting $\hat{A}$

We can see, there are no inner loop dependencies, which means it is a great function to parallelise. The loops for i and j index in the dense matrix are independent of each other, and only one l for each

j, will have the same index value from the CSC, as the i index in $\mathcal{I}$. This means, we can parallelise the `CSCToBatchedDenseMatrices` on batchsize × maxn1 × maxn2 × maxOffset, where maxn1 is the maximum value of n1 in the batch, maxn2 is the maximum value of n2 in the batch, and maxOffset is the max value of offset in the batch. The kernel then looks like this:

```
__global__ void CSCToBatchedDenseMatrices(CSC* d_A, float** d_AHat, int**
    d_PointerI, int** d_PointerJ, int* d_n1, int* d_n2, int maxn1, int maxn2
    , int maxOffset, int batchsize) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < batchsize * maxn1 * maxn2 * maxOffset) {
        int b = (tid / (maxn1 * maxn2 * maxOffset));
        int i = (tid % (maxn1 * maxn2 * maxOffset)) / (maxn2 * maxOffset);
        int j = ((tid % (maxn1 * maxn2 * maxOffset)) % (maxn2 * maxOffset))
    / maxOffset;
        int l = ((tid % (maxn1 * maxn2 * maxOffset)) % (maxn2 * maxOffset))
    % maxOffset;

        int n1 = d_n1[b];
        int n2 = d_n2[b];

        int* I = d_PointerI[b];
        int* J = d_PointerJ[b];
        float* AHat = d_AHat[b];

        if (i < n1 && j < n2) {
            int offset = d_A->offset[J[j]];
            int offsetDiff = d_A->offset[J[j] + 1] - offset;
            if (l < offsetDiff) {
                if (I[i] == d_A->flatRowIndex[l + offset]) {
                    AHat[i * maxn2 + j] += d_A->flatData[l + offset];
                }
            }
        }
    }
}
...
numBlocks = (batchsize * maxn1 * maxn2 * A->m + BLOCKSIZE - 1) / BLOCKSIZE;
CSCToBatchedDenseMatrices<<<numBlocks, BLOCKSIZE>>>(d_A, d_PointerAHat,
    d_PointerI, d_PointerJ, d_n1, d_n2, maxn1, maxn2, A->m, batchsize);
```

Listing 16: Implementation of setting batch $\hat{A}$ in parallel

This kernel gets unique indices for b, i, j and l calculated from the thread ID. A prevoius kernel `setMatrixZero` sets all the indeces of d_AHat to 0.0, and `CSCToBatchedBatchedMatrices` then copies the values with the indices sets from $\mathcal{I}$ and $\mathcal{J}$ into d_AHat after. This ensures we pad all the matrices with zeros, if they do not have the same size, which is important for both the QR decomposition and inverse functions from cuBLAS.
Then we perform the QR decomposition and solve the LS problem. These computations will be discussed in the next sections.

The while-loop is initialised with the stopping conditions tolerance and maxIterations set by the user as input to the `parallelSpai` function. All the computations inside the while-loop is performed with kernels and since there are quite many, we have chosen not to highlight them. They follow the same structure and principles as the kernels explained already and later in the next sections.

The last step of the algorithm is updating the columns of M corresponding to the subproblems in the batch. We do so with the kernel `updateBatchColumnsCSC`. This function currently works on a dense representation of M, which is not ideal. Ideally we would like to implement this function to

work in parallel on a sparse representation of M, but this proved to be difficult and time consuming to implement. For future optimisations, this would be fixed.

5.3.3 The qrBatched function: Computing the QR decomposition of batched matrices

In the sequential version we already implemented the QR decomposition with a batched cuBLAS function and merely sat the number of matrices in the batch to 1. In the parallel version the number of matrices in the batch is the batchsize, which is given to `parallelSpai` and passed on to `qrBatched`. Similarly to the sequential version, we use the cuBLAS function `cublasSgeqrfBatched` [10] to compute the QR decomposition with Householder reflections. In order to obtain Q from this we use algorithm 1 [13], as we did in the sequential implementation. However, this time we will use kernels instead of for-loops in order to make it parallel. We have used the kernels:

- `copyRFromAHat`
- `SetQToIdentity`
- `makeV`
- `matrixMultiplication`
- `computeQvTimesVTransposed`
- `computeQminusQvvt`

The `matrixMultiplication` kernel will be explained in the section concerning the update of the QR decomposition.

5.3.4 The LSProblem function: Solving the least squares problem in parallel

In the parallel version, we of course also have to solve the least squares problem. We call the `LSProblem` function from the `parallelSpai` function and the structure of the function follows step 5 and 6 in the pseudocode. See the structure here:

Algorithm 9: Structure of LSProblem function

Input : cHandle: the cublas handle, A: the sparse matrix, d_PointerQ: the pointer to the Q matrices, d_PointerR: the pointer to the R matrices, d_mHat_k: the pointer to the mHat_k vectors, d_PointerPc: the pointer to the column permutation matrices, d_PointerResidual: the pointer to the residual vectors, d_PointerI: the pointer to the I vectors, d_PointerJ: the pointer to the J vectors, d_n1: the number of rows in A, d_n2: the number of columns in A, k: the index of the column to be added, residualNorm: the norm of the residual, batchsize: the batchsize for the cublas handle

Output: int 0 for success or int 1 for failure

5.1) Compute $\hat{c} = Q^T \hat{e}_k$

Kernel: `SetCHat`

5.2) Compute R_{-1}

Function: `invBatched`

5.3) Compute $\hat{m}_k = R^{-1}\hat{c}$

Kernel: `computeMHat`

Permute $\hat{m}_k$ if necessary

Kernel: `matrixMultiplication`

6) Compute residual

Kernel: `computeResidual`

Kernel: `computeNorm`

First we compute the vector $\hat{c}$ with the kernel `SetCHat`. In the sequential implementation we compute $\hat{c}$ with for-loops as seen below:

```
float* cHat = (float*) malloc(n2 * sizeof(float));
for (int j = 0; j < n2; j++) {
    cHat[j] = 0.0;
    for (int i = 0; i < n1; i++) {
        cHat[j] += Q[i * n1 + j] * e_k[i];
    }
}
```

Listing 17: Sequential implementation of computing $\hat{c}$

This function consists of two nested for loops with an inner loop dependency, and is therefore an ideal target for both parallelisation and optimisation.

```
__global__ void setCHat(float** d_PointerCHat, float** d_PointerQ, int**
    d_PointerI, int* d_n1, int currentBatch, int batchsize, int maxn1) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < maxn1 * maxn1 * batchsize) {
        int b = tid / (maxn1 * maxn1);
        int i = (tid % (maxn1 * maxn1)) / maxn1;
        int j = (tid % (maxn1 * maxn1)) % maxn1;
        int k = currentBatch * batchsize + b;

        int n1 = d_n1[b];

        float* d_cHat = d_PointerCHat[b];
        float* d_Q = d_PointerQ[b];
        int* d_I = d_PointerI[b];

        if (j == 0) {
            d_cHat[i] = 0.0;
        }
        __syncthreads();
```

```
19
20          if (i < n1) {
21              if (k == d_I[i]) {
22                  d_cHat[j] = d_Q[i * maxn1 + j];
23              }
24          }
25      }
26  }
27  ...
28  setCHat<<<numBlocks, BLOCKSIZE>>>(d_PointerCHat, d_PointerQ, d_PointerI,
        d_n1, currentBatch, batchsize, maxn1);
```

Listing 18: Parallel implementation of computing $\hat{c}$

We have translated these for-loops into a kernel, where each thread performs its own computation for it's assigned index. In the sequential version, we multiplied Q with the k'th unit vector. We realised this is unnecessary work, since the multiplication just gives us the k'th row of Q. For the parallel version, we skipped the step with the unit vector, and just checked whether the i'th index in $\mathcal{I}$ matched with the k'th index. If it did, we copied the corresponding index in Q to the j'th index in $\hat{c}$.

In this kernel, we use the value of index b in the array d_n1 twice, and for this reason we save the variable in the kernel for faster memory access, and thus better temporal locality. Consecutive threads also access the memory location right next to each other, when indexing in the arrays. A given thread with index x and it's neighbour with index $x+1$, will have the same values for b and i and then the second thread will have index j one greater than the first thread. Since the data in the arrays are stored in row major fashion in a flat array, all threads should access data right next to each other in memory. For further optimisation of this kernel, we could store the data in the blocks shared memory, for even faster memory access. All our kernels should follow these principles and could all benefit from the usages of per block shared memory.

We then compute the inverse of R and use R^{-1} and $\hat{c}$ to compute the $\hat{m}_k$.

5.3.5 The invBatched function: Inversion on batched matrices

When performing the inversion of R we do the same as in the sequential version and use the cuBLAS function `cublasSgetriBatched` [10], but this time we actually use the batched functionality. The number of matrices in the batch is the batchsize, which is given to `parallelSpai` and passed on to `invBatched`.

5.3.6 The updateQR function: Updating the QR decomposition parallel

As in the sequential version, is the update of the QR decomposition an important part of the SPAI algorithm. We have implemented the sub-steps of step 13 and also step 14 from the pseudocode in the function `updateQR`. This is the structure of the function, which takes quite some arguments:

Algorithm 10: Structure of updateQR function in CUDA

Input : cHandle: cublas handle, d_A: pointer to A in device memory, d_PointerQ: pointer to Q in device memory, d_PointerR: pointer to R in device memory, d_PointerI: pointer to I in device memory, d_PointerJ: pointer to J in device memory, d_PointerSortedJ: pointer to sortedJ in device memory, d_PointerITilde: pointer to ITilde in device memory, d_PointerJTilde: pointer to JTilde in device memory, d_PointerIUnion: pointer to IUnion in device memory, d_PointerJUnion: pointer to JUnion in device memory, d_n1: pointer to n1 in device memory, d_n2: pointer to n2 in device memory, d_n1Tilde: pointer to n1Tilde in device memory, d_n2Tilde: pointer to n2Tilde in device memory, d_n1Union: pointer to n1Union in device memory, d_n2Union: pointer to n2Union in device memory, d_mHat_k: batched mHat_k in device memory, d_PointerMHat_k: pointer to mHat_k in device memory, d_PointerResidual: pointer to residual in device memory, d_residualNorm: pointer to residualNorm in device memory, maxn1: maximum value of n1, maxn2: maximum value of n2, maxn1Tilde: maximum value of n1Tilde, maxn2Tilde: maximum value of n2Tilde, maxn1Union: maximum value of n1Union, maxn2Union: maximum value of n2Union, i: current iteration, batchsize: batchsize

Output: int 0 for success or int 1 for failure

13.1) Create $A(I, \tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$
Kernel: `CSCToBatchedDenseMatrices`
Function: `createPermutationMatrices`
13.2) Compute $\breve{A} = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$
Kernel: `computeABreve`
13.3) Compute $B_1 = \breve{A}(0 : n_2, 0 : n_2)$
Kernel: `SetB1`
13.4) Compute $B_2 = \breve{A}(n_2 + 1 : n_1, 0 : \tilde{n_2})$ above $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$
Kernel: `SetB2`
13.5) Do QR decomposition of B_2
Function: `qrBatched`
13.6) Compute Q and R
Kernel: `SetFirstMatrix`
Kernel: `SetSecondMAtrix`
Kernel: `MatrixMultiplication`
Kernel: `SetUnsortedR`
13.7) Solve the augmented LS problem for $\hat{m}_k$
Kernel: `LSProblem`
Kernel: `permuteJ`
14) Compute the residual
Kernel: `computeResidual`
Kernel: `computeNorm`

In the first step of updating the QR decomposition, we create $A(I, \tilde{\mathcal{J}})$ and $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ with the kernel `CSCToBatchedDenseMatrices` already described. We do so after using the kernel `floatDeviceToDevicePointerKernel` to obtain the pointer arrays, that holds the pointers that point to the beginning of each d_AIJTilde and d_AITildeJTilde matrix respectively.

In this function, aswell as in many others, we use the kernel `matrixMultiplication`. The sequential implementation of matrix multiplication consists of three nested for-loops.

```
// compute unsortedQ = firstMatrix * secondMatrix
float* unsortedQ = (float*) malloc(n1Union * n1Union * sizeof(float));
for (int i = 0; i < n1Union; i++) {
    for (int j = 0; j < n1Union; j++) {
        unsortedQ[i * n1Union + j] = 0.0;
        for (int k = 0; k < n1Union; k++) {
            unsortedQ[i * n1Union + j] += firstMatrix[i * n1Union + k] *
   secondMatrix[k * n1Union + j];
        }
    }
}
```

Listing 19: Sequential implementation of matrix multiplication

Looking at the sequential code, we can identify that the innermost loop changes the values we have to multiply for each iteration, meaning we have an innerlopp dependency. We want to save the sum of these multiplication at the index of the i'th row and j'th column of the result matrix. Since only the innermost loop changes, we can parallelise the two outer loops. This means we can parallelise on the dimensions of the result matrix. The `matrixMultiplication` kernel looks like the following, where we have also parallelised on the batchsize:

```
__global__ void matrixMultiplication (float** d_PointerA, float**
   d_PointerB, float** d_PointerC, int* d_dim1, int* d_dim2, int* d_dim3,
   int maxdim1, int maxdim2, int maxdim3, int batchsize) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < batchsize * maxdim1 * maxdim3) {
        int b = tid / (maxdim1 * maxdim3);
        int i = (tid % (maxdim1 * maxdim3)) / maxdim3;
        int j = (tid % (maxdim1 * maxdim3)) % maxdim3;

        int dim1;
        int dim2;
        int dim3;

        if (d_dim1 == NULL) {
            dim1 = maxdim1;
        } else {
            dim1 = d_dim1[b];
        }

        if (d_dim2 == NULL) {
            dim2 = maxdim2;
        } else {
            dim2 = d_dim2[b];
        }

        if (d_dim3 == NULL) {
            dim3 = maxdim3;
        } else {
            dim3 = d_dim3[b];
        }

```

```
        float* d_A = d_PointerA[b];
        float* d_B = d_PointerB[b];
        float* d_C = d_PointerC[b];

        if (i < dim1 && j < dim3) {
            float sum = 0.0;
            for (int k = 0; k < dim2; k++) {
                sum += d_A[i * maxdim2 + k] * d_B[k * maxdim3 + j];
            }
            d_C[i * maxdim3 + j] = sum;
        } else {
            d_C[i * maxdim3 + j] = 0.0;
        }
    }
}
...
numBlocks = (batchsize * maxn1Union * maxn1Union + BLOCKSIZE - 1) /
    BLOCKSIZE;
matrixMultiplication<<<numBlocks, BLOCKSIZE>>>(d_PointerFirstMatrix,
    d_PointerSecondMatrix, d_PointerUnsortedQ, d_n1Union, d_n1Union,
    d_n1Union, maxn1Union, maxn1Union, maxn1Union, batchsize);
```

Listing 20: Parallel implementation of matrix multiplication

We get the unique set of the indices b, i and j, and then run a for-loop inside the kernel computing the sum of the multiplication of the i'th row of the first matrix, with the j'th column of the second matrix. This value is then inserted in the result matrix at the indices $[i, j]$.

The rest of the steps are each computed with their own kernels. All of the kernels are basically the sequential for-loops from the C version translated into kernels in the way we have shown in the previous sections.

5.3.7 Shortcomings of the parallel implementation

Our parallel implementation is currently not working for all matrix size and batchsizes. When running the code on a large matrix we get the CUDA error `cudaErrorMisalignedAddress` or `cudaErrorIllegalAddress`. We have identified, that it happens after a call to `setUnsortedR`. We have done the following in our code the localise the error:

```
cudaError_t test = cudaDeviceSynchronize();
printf("test1 = %d\n", test);

numBlocks = (batchsize * maxn1Union * maxn2Union + BLOCKSIZE - 1) /
    BLOCKSIZE;
setUnsortedR<<<numBlocks, BLOCKSIZE>>>(d_PointerUnsortedR, d_PointerR,
    d_PointerB1, d_PointerB2R, d_n1, d_n1Union, d_n2, d_n2Union, d_n2Tilde,
    maxn1Union, maxn2, maxn2Tilde, maxn2Union, batchsize);

test = cudaDeviceSynchronize();
printf("test2 = %d\n", test);
```

Listing 21: Code for finding the error(s) in our code

The CUDA function `cudaDeviceSynchronize()` ensures the previous kernels are done doing their work, and it returns 0, if the kernels succeeded, and another number if there is a problem. When running our code, we get the output:

```
test1 = 0
test2 = 716
```

or

```
test1 = 0
test2 = 700
```

This means that the kernels we run, before the first `cudaDeviceSynchronize()` succeeded in their computation, while the kernel `setUnsortedR` fails. The first error indicates that somehow the data in the arrays have been shifted, so they are no longer aligned with how float data should be aligned. The second error suggests that the device encountered a load or store instruction from or to an invalid memory address. We have done our best to ensure the kernel is correct. The kernel looks like this:

```
__global__ void setUnsortedR(float** d_PointerUnsortedR, float** d_PointerR
    , float** d_PointerB1, float** d_PointerB2R, int* d_n1, int* d_n1Union,
    int* d_n2, int* d_n2Union, int* d_n2Tilde, int maxn1Union, int maxn2,
    int maxn2Tilde, int maxn2Union, int batchsize) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid < batchsize * maxn1Union * maxn2Union) {
        int b = tid / (maxn1Union * maxn2Union);
        int i = (tid % (maxn1Union * maxn2Union)) / maxn2Union;
        int j = (tid % (maxn1Union * maxn2Union)) % maxn2Union;

        int n1 = d_n1[b];
        int n1Union = d_n1Union[b];
        int n2 = d_n2[b];
        int n2Union = d_n2Union[b];
        int n2Tilde = d_n2Tilde[b];

        float* d_UnsortedR = d_PointerUnsortedR[b];
        float* d_R = d_PointerR[b];
        float* d_B1 = d_PointerB1[b];
        float* d_B2R = d_PointerB2R[b];

        if (i < maxn1Union && j < maxn2Union) {
            if (i < n2 && j < n2) {
                d_UnsortedR[i * maxn2Union + j] = d_R[i * maxn2 + j];
            } else if (i < n2 && j < n2Union && j > n2 - 1) {
                d_UnsortedR[i * maxn2Union + j] = d_B1[i * maxn2Tilde + j -
    n2];
            } else if (i < n2Union && j < n2Union && j > n2 - 1) {
                d_UnsortedR[i * maxn2Union + j] = d_B2R[(i - n2) *
    maxn2Tilde + j - n2];
            } else {
                d_UnsortedR[i * maxn2Union + j] = 0.0;
            }
        }
    }
}
```

Listing 22: Code for the setUnsortedR kernel

In this kernel we update R with the values from B_1 and R_B. The new R has the size maxn1Union × maxn2Union, the old R has the size maxn1 × maxn2, though we only need the values from R_1, so we only take the values from the top of R with the size maxn2 × maxn2. B_1 has the size maxn2 × maxn2Tilde and R_B has the size maxn1Tilde × maxn2Tilde, where we again only need the top of size maxn2Tilde × maxn2Tilde. The matrices in the batches, might have different sizes, but are all stored in the top left corner, with padded zeros on the left and on the bottom. The matrices are stacked the following way to get the new R:
The indexing is split up in 4 parts:

First if statement is where we copy the old R to the new R. This is done by indexing with the dimensions of old R. Since we want to keep the padding on the right side and on the bottom,

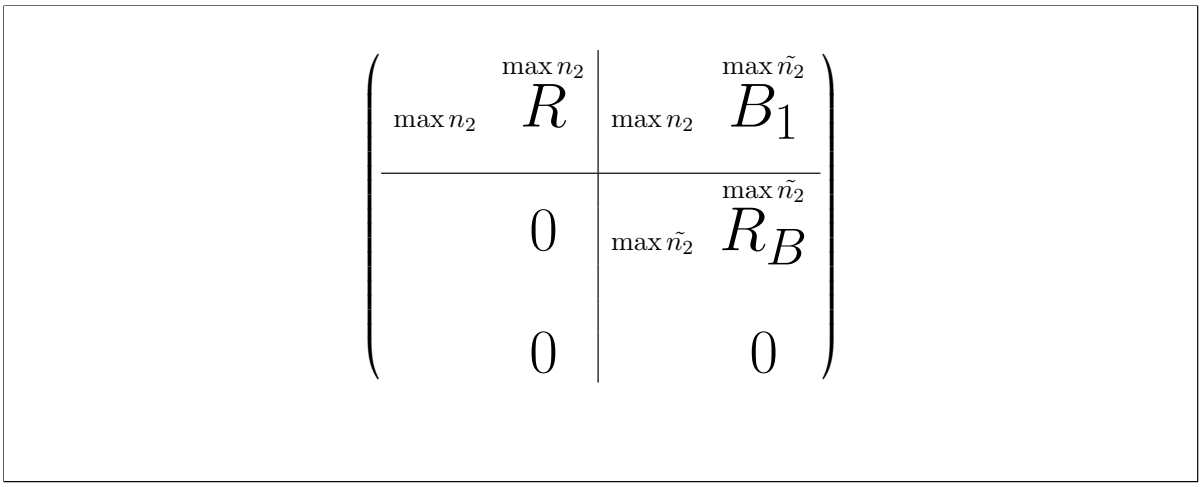

Figure 8: The new R matrix constructed from the old R, B_1 and R_B

we index with the dimensions of the matrix in the specific batch. This copies the values from old R without the zeros to the new R.

Second if statement is where we copy B_1 to new R. As we can see on figure 8 B_1 is on the right side of old R with the same dimension on the columnn axis. With the if statement we ensure we index correctly in new R. The first j falling into this if statement will be the number following the last number from the previous if statement. In B_1 we index with i and $j - n_2$, since the j is shifted n_2 times to the right in respects to B_1.

Third if statement is where we copy R_B to thge new R. R_B has the same dimension on the x-axis as B_1 and is maxn2Tilde on the y-axis. It is placed right under B_1. The if statement ensures we index correctly in new R by only falling into this case, if i is under n2Union. If i is smaller than n_2 i would fall under the previous if statement. Since i and j are shifted by n_2 to the right and down, we subtract n_2 from i and j when indexing in B_1

Fourth if statement is where we to add the padding zeros. If we get to this point, there are no more values to add, so we just add zeros to the indices.

Since we assume the function `setUnsortedR` works correctly, we suspect the error lies at another point in the code. Most likely a place, where we copy data to either B_1 or R_B. We have yet to identify exactly where in the code this happens.

Batchsize vs. matrix size
When testing our parallel implementation, we have figured out, that our program fails when either the batch size or the matrix size gets too big. We suspect we start running into issues, when we have to use more threads than, there are present in one block. When the kernels start using more blocks than one, the computational behaviour changes. Within a kernel, it is only possible to synchronise the threads in one block, meaning other blocks might be done with their computations of a given step. This can lead to those blocks starting computation of the next steps. If the next step involves reading or writing to memory, that another block on another computation step is reading or writing to at the same time, undefined behaviour might happen, and we cannot be sure what the output will be. This is potentially getting the wrong values or accessing illegal memory.

The GPU computations and the CPU computations are asynchronous, meaning, that one of them might jump ahead of the other in the code, if they are not explicitly instructed not to with e.g. the

CUDA function `cudaDeviceSynchronize`. Since we do not use more than the default stream in CUDA this should not be an issue for us, since kernels will wait for the execution of the previous kernel to finish before starting. Even thought this is the case, an examination of this is on the todo list of future work, to ensure our code works properly. The same is the case for asynchronous behaviour in our kernels.

5.4 Running the implementations

The python implementation in run in the terminal either directly or trough the `test.py` file, if one wants to see the time and accuracy.

```
$ python3 SPAI.py
$ python3 test.py
```

The C implementation is written in CUDA header files and the executable file `testSpai.cu` is run with the help of the `Makefile`.

```
# compile
$ make compile

# run
$ make run

# run tests of speed
$ make test

# compile and run
$ make
```

The CUDA implementation is also written in CUDA header files and the executable file `testSpai.cu` is run with the help of the `Makefile`.

```
# compile
$ make compile

# run
$ make run

# run tests of speed
$ make test

# compile and run
$ make
```

Both the C and the CUDA implementations are tested on DIKUs Futhark servers. The experiments we have performed on the servers will be shown in section 6 concerning the numerical experiments.

6 Numerical experiments

In this section, we will show the results of the numerical experiments, we have performed on our implementations. We will compare the accuracy of our three implementations with different parameters. We will compare the run time of our implementations with the library functions from Scipy in Python and cuSOLVER in CUDA.

We have chosen to test the implementations on randomised matrices, that we have generated from a given size and sparsity degree. One could have made the numerical experiments more relevant by using data from problems coming from scientific and industrial applications. Grote and Huckle use data from problems such as imcompressible flow in a pressure driven pipe and a landing hydrofoil airplane model [1]. This would have made the experiments more relevant for real life use.

We have run the C and CUDA experiments on the Futhark servers provided by DIKU [11].

Device name:	NVIDIA A100-PCIE-40GB
Number of hardware threads:	221184
Max block size:	1024
Shared memory size:	49152

The Python experiments were executed on a desktop.

Processor:	AMD Ryzen 5 5600X, 3.7 GHz
Memory:	16 GB Vengeance LPX - DDR4 3000 MHz

6.1 Measuring the accuracy

For our own working implementations we have measured the accuracy. We have calculated the error, which we compute as the norm of identity matrix I subtracted from the result M multiplied with the input matrix A.

$$\text{Error} = \|AM - I\| \tag{72}$$

We have used the same parameters for every test and only changed the size and the sparsity of A in order to get the most precise and also broad results. The parameters have been set to

$$\text{Tolerance} = 0.01, \quad \text{Max iterations} = n - 1, \quad \text{Smallest indices} = 1$$

One could have also changed these 3 parameters in order to complete a more wide experiment, but since our current implementation only works for these arguments we have chosen not to change them. We have tested on

$$\text{Sparsity of } A = \{0.1, 0.3, 0.5\}$$
$$\text{Size of } A = \{10 \times 10, 100 \times 100, 1000 \times 1000\}$$

We could not test the implementations for sizes greater than 1000×1000 since the Python version was killed, and the C version took so long that we did not obtain a result. The parallel version only works for 10×10 matrices.

Here are our results:

	10 x 10	100 x 100	1000 x 1000
Sequential Python implementation	A is singular	1.28E-13	3.21E-12
Sequential C implementation	A is singular	3.28E-04	2.87E-02
Parallel CUDA implementation	A is singular	-	-

Figure 9: Testing the accuracy with parameters: Sparsity of A = 0.1, Tolerance = 0.01, Max iterations = n - 1 and s = 1

	10 x 10	100 x 100	1000 x 1000
Sequential Python implementation	2.06E-14	5.72E-14	3.08E-12
Sequential C implementation	8.86E-10	1.35E-03	5.36E-02
Parallel CUDA implementation	3.67E-11	-	-

Figure 10: Testing the accuracy with parameters: Sparsity of A = 0.3, Tolerance = 0.01, Max iterations = n - 1 and s = 1

	10 x 10	100 x 100	1000 x 1000
Sequential Python implementation	2.86E-15	2.03E-13	2.12E-12
Sequential C implementation	8.39E-11	4.44E-04	1.98E-02
Parallel CUDA implementation	3.98E-11	-	-

Figure 11: Testing the accuracy with parameters: Sparsity of A = 0.5, Tolerance = 0.01, Max iterations = n - 1 and s = 1

6.2 Measuring the run time

We want to find the run time of the different implementations in order to sufficiently compare the efficiency of the implementations. We have calculated the time used with library functions for time measurement. The results are presented in seconds.

Similarly to the accuracy tests, we have used the same parameters for every test and only changed the size and the sparsity of A. The parameters have been set to

$$\text{Tolerance} = 0.01, \quad \text{Max iterations} = n - 1, \quad \text{Smallest indices} = 1$$

and

$$\text{Sparsity of } A = \{0.1, 0.3, 0.5\}$$
$$\text{Size of } A = \{10 \times 10, 100 \times 100, 1000 \times 1000, 10000 \times 10000\}$$

We have compared our own working Python and C implementations with the Scipy function `scipy.sparse.linalg.inv(A)` for finding the exact inverse of a sparse matrix.

We have also compared with the cuSOLVER function `cusolverDnSgetrf` function that finds the exact inverse of triangular matrix. Since A is not triangular, we have to perform LU decomposition of A in order to use the function to find the inverse of the upper triangular matrix U and lower triangular matrix L. This will of course add some extra run time for the function.

Here are our results:

	10 x 10	100 x 100	1000 x 1000	10000 x 10000
Scipy implementation	A is singular	0.014879	0.735263	919.821114
Sequential Python implementation	A is singular	13.778171	7391.510661	-
CuSOLVER implementation	A is singular	0.002633	0.012987	3.350172
Sequential C implementation	A is singular	69.074035	5023.625312	-

Figure 12: Testing the speed with parameters: Sparsity of A = 0.1, Tolerance = 0.01, Max iterations = n - 1 and s = 1

	10 x 10	100 x 100	1000 x 1000	10000 x 10000
Scipy implementation	0.002563	0.016799	0.740422	806.18316
Sequential Python implementation	0.106887	12.471273	6602.605517	-
CuSOLVER implementation	0.002086	0.002700	0.013012	3.498521
Sequential C implementation	0.018003	92.412611	5208.116394	-

Figure 13: Testing the speed with parameters: Sparsity of A = 0.3, Tolerance = 0.01, Max iterations = n - 1 and s = 1

	10 x 10	100 x 100	1000 x 1000	10000 x 10000
Scipy implementation	0.001761	0.017183	0.7267112	811.357949
Sequential Python implementation	0.078199	12.866873	7125.50547	-
CuSOLVER implementation	0.002052	0.002724	0.013023	3.951132
Sequential C implementation	0.021173	103.863722	6198.732912	-

Figure 14: Testing the speed with parameters: Sparsity of A = 0.5, Tolerance = 0.01, Max iterations = n - 1 and s = 1

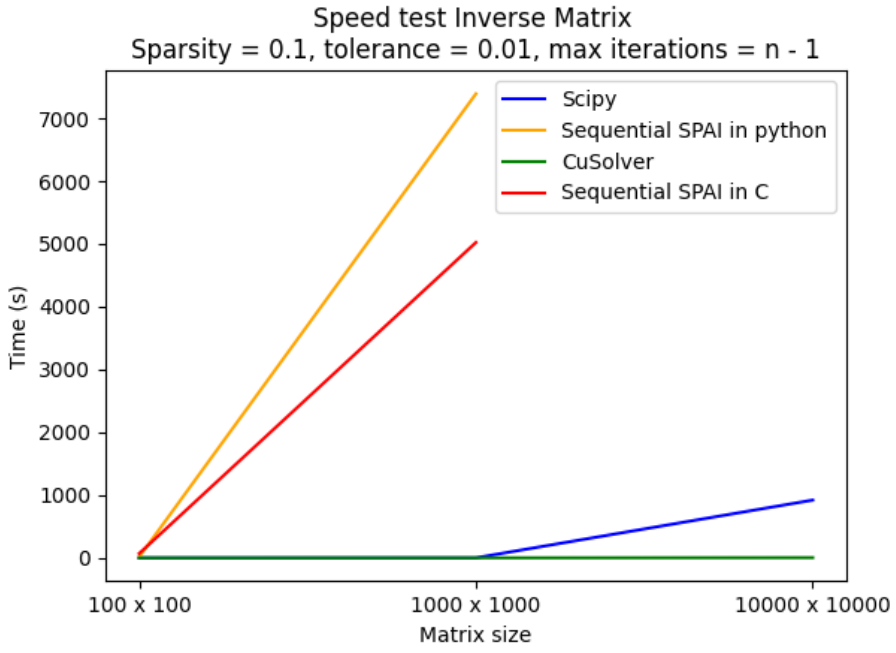

Figure 15: Testing the speed with parameters: Sparsity of A = 0.1, Tolerance = 0.01, Max iterations = n - 1 and s = 1

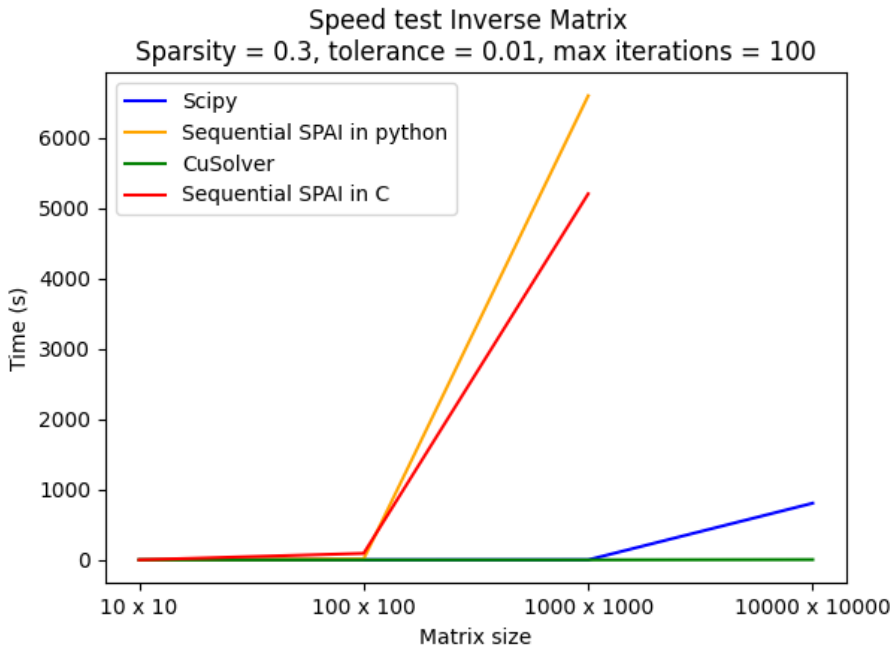

Figure 16: Testing the speed with parameters: Sparsity of A = 0.3, Tolerance = 0.01, Max iterations = n - 1 and s = 1

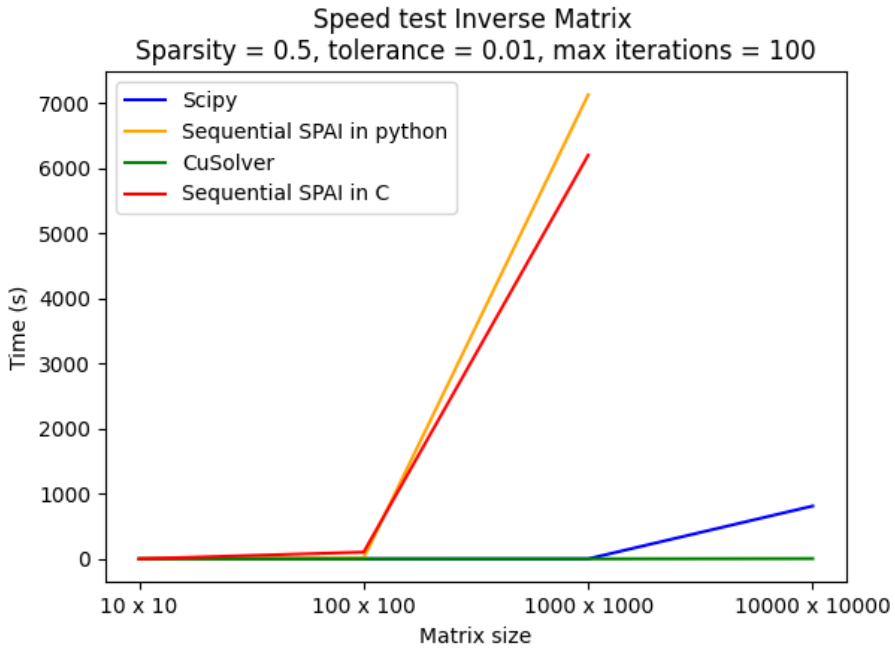

Figure 17: Testing the speed with parameters: Sparsity of A = 0.5, Tolerance = 0.01, Max iterations = n - 1 and s = 1

6.3 Testing the CUDA kernels

We conducted experiments to compare a parallel implementation running on the GPU with a sequential implementation running on the CPU. However, due to limitations in our parallel CUDA implementation for large matrices, we were unable to obtain sufficient test results for direct comparison. Instead, we focused on individual CUDA kernels within our parallel implementation.

We measured the run time of a CUDA kernel executed on the GPU and compared it with the run time of the corresponding sequential code executed on the CPU. We performed the experiments on the kernels:

- `MatrixMultiplication`
- `SetSecondMatrix`

We selected these two kernels specifically because they encompass essential and frequently used functionalities within our implementation. The `MatrixMultiplication` kernel, as its name suggests, performs matrix multiplication, a crucial operation that is utilised multiple times throughout our implementation. The `SetSecondMatrix` kernel sets the second matrix used in the matrix multiplication, which is essential for computing Q. It serves as a representative example of how we set matrices, a task that is performed multiple times in our implementation.

Since these kernels perform computations on dense submatrices in the SPAI algorithm, we have tested on matrices of

$$\text{sparsity} = 1.0 \text{ and sizes} = \{10 \times 10, 100 \times 100, 500 \times 500, 1000 \times 1000, 5000 \times 5000\}.$$

Here are the results:

`matrixMultiplication`	10 x 10	100 x 100	500 x 500	1000 x 1000	5000 x 5000
Sequential code	0.000011	0.006504	0.746897	4.025718	1031.439322
Kernel	0.000005	0.000004	0.000004	0.000005	0.000006

Figure 18: Run times in seconds of the kernel `matrixMultiplication` and the corresponding sequential code

`SetSecondMatrix`	10 x 10	100 x 100	500 x 500	1000 x 1000	5000 x 5000
Sequential code	0.000007	0.000083	0.00193	0.007267	0.173633
Kernel	0.000009	0.000011	0.000011	0.00002	0.000183

Figure 19: Run times in seconds of the kernel `SetSecondMatrix` and the corresponding sequential code

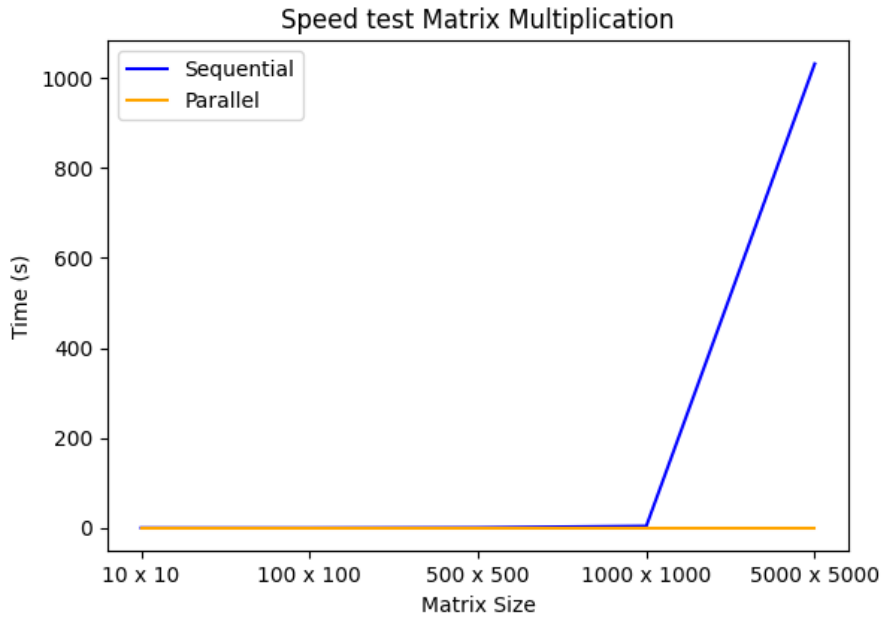

Figure 20: Testing the speed of `matrixMultiplication` on dense matrices

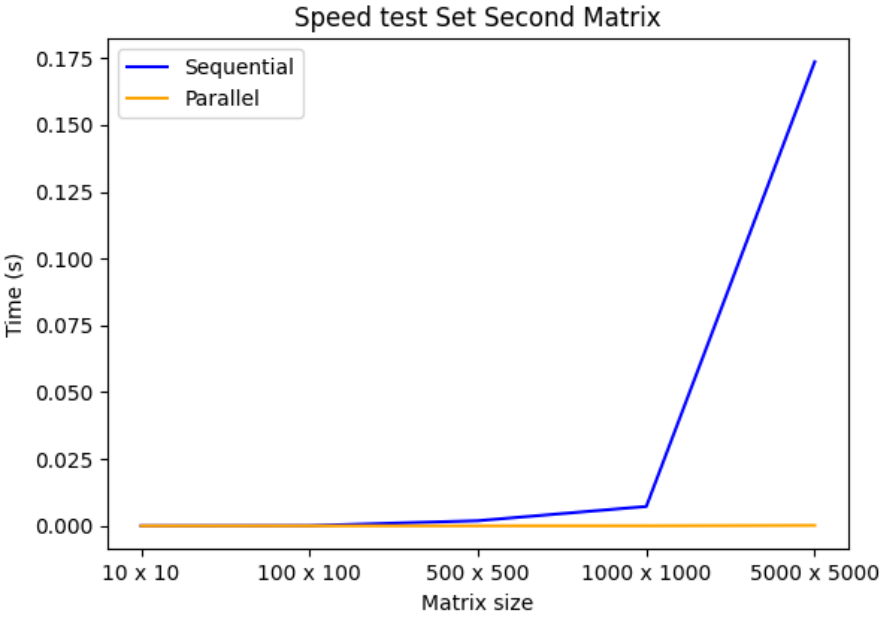

Figure 21: Testing the speed of `SetSecondMatrix` on dense matrices

We conducted measurements to determine the amount of gigabytes per second (GB/s) utilised by the kernels for various matrix sizes. Additionally, we calculated the percentage of peak performance of the GPU that these kernels were able to achieve. The peak performance of DIKUs Futhark servers is 1600 GB/s.

Here are the results:

`matrixMultiplication`	10 x 10	100 x 100	500 x 500	1000 x 1000	5000 x 5000
GB/s	0.16	20	500	1600	1600
% of peak performance	0.01%	1.25%	31.25%	100.00%	100.00%

Figure 22: GB/s used by the kernel `matrixMultiplication` and percentage of the peak performance

`SetSecondMatrix`	10 x 10	100 x 100	500 x 500	1000 x 1000	5000 x 5000
GB/s	00.09	7.27	181.82	400	1092.9
% of peak performance	0.00%	0.45%	11.36%	25.00%	68.31%

Figure 23: GB/s used by the kernel `SetSecondMatrix` and percentage of the peak performance

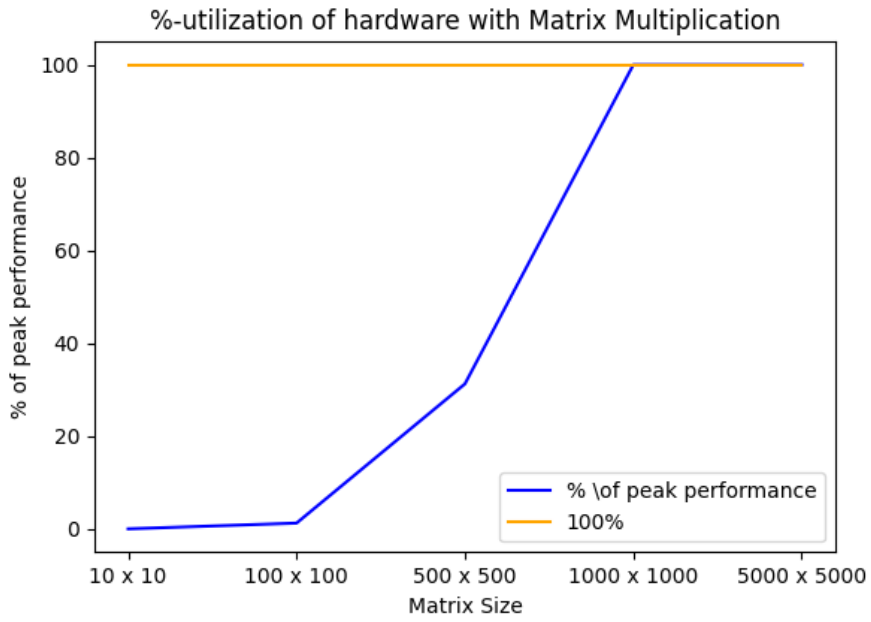

Figure 24: % of peak performance when running `matrixMultiplication` on GPU

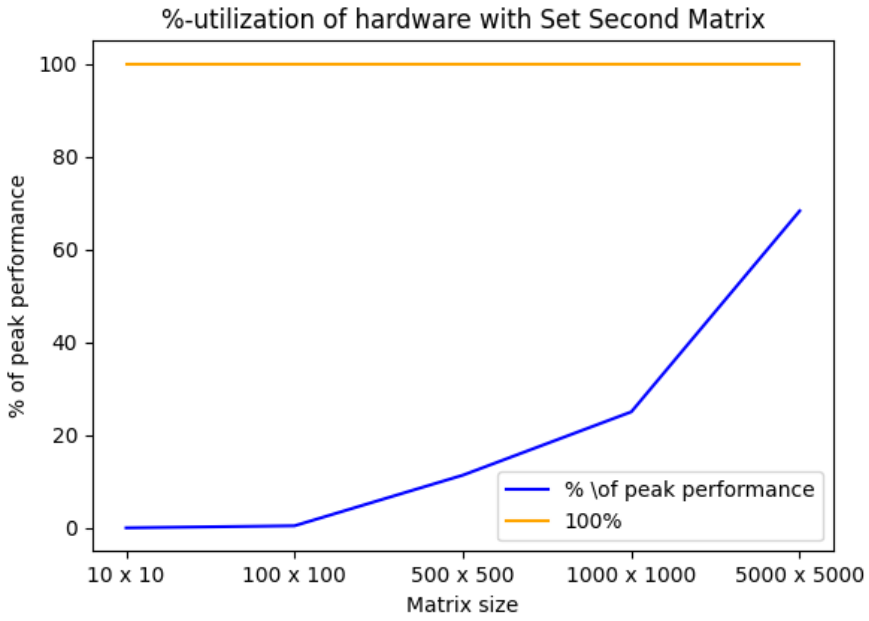

Figure 25: % of peak performance when running `SetSecondMatrix` on GPU

7 Discussion

In this section we will use our results from the numerical experiments to compare the accuracy and speed of our sequential and parallel implementations. We will compare the run time of our working Python and C implementations with the library functions for finding the exact inverse.

7.1 Comparison of the accuracy of our implementations

The results from the numerical experiments show that the norm of $AM - I$ is very close to zero for all of the sizes and sparsity degrees of the matrices. This means that the resulting approximate inverses are very close to be the exact inverse of the input matrix A.

For matrices of size 10×10 and sparsity degree 0.1, the matrices are singular basically all the time. That is because there is a very large possibility of the determinant being zero due to zero rows or columns, when the matrix is both small and very sparse.

The Python version shows the smallest error values and thus the closest approximations. However when comparing number such e.g. 2.06E-14 and 8.86E-10, we should keep in mind that these are very small decimal numbers. For matrices of size 100×100 and 1000×1000, we observe some more noticeable errors in the results of the C implementation. However, the values such as 5.36E-02 are still very close zero and within the set tolerance, and thus accepted.

As our parallel implementation does not successfully find the approximate inverse for large matrices have we only obtained results from a 10×10 matrix. The results are very small decimal values such as 3.67E-11, which are within the accepted tolerance.

7.2 Comparison of the run time of our implementations and library functions

We conducted experiments to determine the run time of our sequential implementations as well as the functions in Scipy and cuSOLVER for finding the exact inverse. The results clearly indicate that the library functions outperformed our own implementations in terms of speed.

Our C and Python implementations performed well, when dealing with small matrices. However, when we tested them on matrices of size 1000×1000, the run time exceeded 5000 seconds, which is approximately 1.5 hours. For matrices of size 1000×1000, the Python test was terminated by the terminal, and the C version did not converge even after running for an entire day. We anticipated that the Python version would be slow and unsuitable for large matrices, as the purpose was not efficiency, but rather serving as prototypes for subsequent implementations. The sequential C version was also not designed for efficiency, but rather as a stepping stone in the process of implementing the parallel version. Nonetheless, optimisations could be applied to reduce the running time and enable the implementation to handle larger matrices within an acceptable timeframe.

The results highlight that the cuSOLVER function for finding the exact inverse significantly outperformed our own implementations in terms of run time. For large matrices, it surpassed our implementations by thousands of seconds. For instance, when dealing with a 1000×1000 matrix, the C implementation took 6198 seconds, while cuSOLVER completed the calculations in a mere 0.01 second. Even for a 1000×1000 matrix, cuSOLVER produced a result in under 4 seconds. Scipy's function for finding the exact inverse also performed well, although it took slightly more time for very large matrices e.g. it required 811 seconds for a 1000×1000 matrix.

In theory, implementations using the SPAI algorithm, which finds an approximate inverse, should be faster than those finding the exact inverse for very large and sparse matrices. However, our SPAI implementations are currently unable to produce satisfactory results for matrices that are sufficiently large. The library functions in Scipy and cuSOLVER are state-of-the-art implementations constantly

being developed to meet optimal standards. Consequently, it is difficult to compete with their run times. The graphs clearly demonstrate the significantly higher run times of our sequential implementations compared to the library implementations.

If our parallel version had successfully worked for large matrices, it would have likely resulted in a significantly faster run time for large matrices than the sequential versions.

7.3 Comparison of run time and utilisation of the GPU by the CUDA kernels

We specifically chose two GPU kernels for evaluation and compared their performance against the corresponding sequential CPU code. The test results for the numerical experiments conclusively establish the superior performance of the parallel GPU version over the sequential CPU counterpart.

The `matrixMultiplication` kernel performs matrix multiplication between two matrices of dimensions $dim1 \times dim2$ and $dim2 \times dim3$, respectively. When using for-loops, this operation consumes significant computational power and exhibits a time complexity of $O(dim1 \times dim2 \times dim3)$. However, through parallelisation, we can reduce the runtime to $O(dim3)$ by traversing only the inner dimensions of the matrices and computing entries concurrently using threads. This causes the theoretical run time to be much faster.

The test results show that also the actual run time is much faster. For a small matrix the sequential version can keep up, due to the overhead of creating the kernel. However, already for a matrix of size 100×100, the parallel version is 8 times faster. For a matrix of size 5000 x 5000 the sequential version takes 1031 seconds, while the parallel version takes 5 microseconds. It can clearly be observed by the graph in the numerical experiment section that the sequential code is far slower for large matrices.

The kernel `SetSecondMatrix` creates a matrix consisting of the identity matrix in the upper left corner and the already computed matrix Q_B in the lower right corner. The sequential code uses two for-loops to set the values in a matrix of size $dim1 \times dim2$ and thus has the time complexity $O(dim1 \times dim2)$. In contrast, the kernel avoids any for-loops and assigns a thread to each entry of the result matrix, enabling parallel computation with a constant time complexity of $O(1)$.

The experimental results not only confirm the theoretical improvement but also demonstrate faster actual run times. As the matrix size increases, the superiority of the parallel version becomes more pronounced. It is worth mentioning that for a 10×10 matrix, the sequential version performs marginally better by 2 microseconds because of the overhead of kernel creation.

The experiments have tested how many gigabytes per seconds the kernels use. We have calculated the percentage-wise utilisation of the peak performance of the servers, which is 1600 GB/s. The results show that the greater the size of the matrix, the greater the utilisation of the server. For the kernel `SetSecondMatrix` we utilise 68.31% for the largest matrix. For `matrixMultiplication` we already reach the peak performance for at matrix of size 1000×1000.

7.4 Advantages of parallelism

Parallelism on the GPU offers numerous advantages in computer science, particularly in the field of parallel implementations. GPUs are equipped with hundreds or even thousands of processing cores, enabling massively parallel execution of tasks. This immense computational power surpasses traditional CPUs, making GPUs highly suitable for computationally intensive applications.

Parallel algorithms can greatly benefit from GPU acceleration. By dividing algorithms into smaller tasks that can be executed simultaneously, GPUs enable parallel execution and result in significant speedups compared to sequential CPU execution.

One significant advantage of GPU parallelism is its ability to expedite data processing. GPUs excel at parallel data handling, making them efficient in managing large datasets. This advantage proves

invaluable in fields such as data analytics, machine learning, and scientific simulations, where the rapid processing of substantial amounts of data is crucial.

In addition to their computational prowess, GPUs have a long-standing association with graphics rendering and visualization. They were initially designed for graphics processing and excel at rendering complex visual scenes in real-time. This makes GPUs indispensable for applications like video games, virtual reality, computer-aided design, and scientific visualization.

Moreover, the versatility of GPUs extends beyond graphics processing. General-purpose computing on GPUs allows developers to leverage the parallel computing capabilities of GPUs for a wide range of applications e.g. an inherently parallel algorithm such as the SPAI preconditioner. This unlocks their potential in various scientific, engineering, and data-intensive tasks beyond traditional graphics-related domains.

Energy efficiency is another notable advantage of GPU parallelism. GPUs can achieve higher performance per watt compared to traditional CPUs, resulting in improved energy efficiency. This aspect is particularly relevant for applications that require intensive parallel processing, as GPUs can deliver higher performance while consuming less power.

7.5 Related works

Sparse Approximate Inverse preconditioners are under constant development and several theses has been dedicated to developing more general and efficient algorithms for computing such.

This thesis is primarily based on the work by Grote and Huckle, whom with their article *Parallel Preconditioning with Sparse Approximate Inverses* presented the original SPAI algorithm [1]. The article explains the SPAI algorithm and provides the pseudocode, which inspired our pseudocode.

However, in favour of brevity details are left out of the work. The article leaves out essential steps in the updating of the QR decomposition; the permutation of Q and R and how to apply the Householder QR decomposition. These parts is elaborated on in our thesis, and inspiration for such has been found in Kallischko [2] and Sedlaceks [3] works. The work of Grote and Huckle dates to 1995 and the library functions and hardware their FORTRAN implementation is run with is now outdated.

Kallischko's dissertation *Modified Sparse Approximate Inverses (MSPAI) for Parallel Preconditioning* presents a more detailed version of the SPAI algorithm and goes into depth with the updating of the QR decomposition [2]. Our algorithm for updating the QR decomposition is inspired by Kallischko and their use of permutation matrices to permute Q, R and $\hat{m_k}$ for use in the next iteration. They also expand on the explanation of the Householder QR decomposition.

However, besides explaining the SPAI algorithm more in depth, they present a modified version of SPAI (MSPAI). MSPAI is modified in order to gain generality. They extend the idea from a $n \times n$ matrix to rectangular $m \times n$ matrices. MSPAI also performs automatic pattern updates, and it thus not restricted to a specific sparsity pattern. They also add information about probing in the form of probing vector augmented with the input matrix. Besides MSPAI, they also present FSPAI, which is a factorised variant of the algorithm that works for symmetric, positive and definite matrices.

In the dissertation *Sparse Approximate Inverses for Preconditioning, Smoothing, and Regularization* Sedlacek accounts for both the SPAI, MSPAI and FSPAI algorithms and develops further on the MSPAI and FSPAI [3]. They present a multi-step variant of the MSPAI algorithm (MMSPAI), which is based on permutations and recursion. They also present a blocked version of the FSPAI algorithm (BFSPAI) and find results that indicate the efficiency of block sparse approximate inverse preconditioners.

One of the newest additions to the area of SPAI preconditioners is the HeuriSPAI algorithm presented

by Gao, Chu and Wand in the paper *HeuriSPAI: a heuristic sparse approximate inverse preconditioning algorithm on GPU* [8]. Compared to the dynamic SPAI preconditioning algorithm in Grote and Huckle (1997) the proposed heuristic SPAI algorithm has two main differences: (1) a heuristic method is proposed to give potential candidate indices and (2) multiple while-loop conditions, that should better maintain the sparsity level of the preconditioner [8]. They solve a problem existing dynamic SPAI preconditioners had with encountering out-of-memory errors for large matrices and validate the effectiveness with a highly parallel experiments of the HeuriSPAI algorithm [8].

8 Conclusion

This thesis focuses on the Sparse Approximate Inverse (SPAI) preconditioner, a method used to calculate an approximate inverse of large and sparse matrices. The SPAI algorithm iteratively constructs a sparse approximation of the inverse matrix, column by column, by minimising the norm. Each iteration updates the given sparsity pattern, while preserving the sparsity of the original matrix. As a result, the computational workload depends on the number of nonzero elements rather than the size of the matrix.

The SPAI algorithm is inherently parallel, as it divides the matrix into subproblems and solves the least squares problem independently for each column. The original SPAI preconditioner, introduced by Grote and Huckle [1], lacks important algorithmic details. In this thesis, we provide a comprehensive explanation of the SPAI algorithm, including the use of QR decomposition with Householder reflections for optimisation and permutations to ensure accuracy. Works by Kallischko [2] and Sedlacek [3] have inspired these aspects.

Our ultimate goal was to implement a parallel version of the SPAI algorithm for GPU execution, but for testing purposes we implemented a sequential prototype in Python to begin with. Subsequently, we developed a sequential version in C, utilising cuBLAS library functions for QR decomposition and inverse calculations of R, to prepare for the parallel implementation. Finally, we implemented the parallel version using CUDA kernels, but the parallel version only works sufficiently for small matrices. We verified the correctness of the kernels when implementing them by comparing against the sequential version step by step.

We conducted experiments on randomised matrices of varying sizes and sparsity levels to measure the accuracy and speed of our implementations. The results demonstrate the accuracy our SPAI implementations. We compared the runtime of our Pyhton and C SPAI implementations with Scipy and cuSOLVERs functions for exact inverse computation. The results show that library functions exhibit superior runtimes to the sequential SPAI implementations. This was expected as our implementations were meant as prototypes and not efficient implementations for practical use.

In theory, the parallel SPAI implementation using CUDA kernels should achieve a fast run time for large and sparse matrices. Since our implementation did not work sufficiently for large matrices, we tested individual kernels. The results show that the kernels executed on the GPU exhibits superior run time to the corresponding sequential code running on the CPU. Future work could involve optimising the implementation for large matrices and thus making it relevant for real-life applications.

We conclude, the Sparse Approximate Inverse preconditioner is an efficient algorithm for calculating the approximate inverse of large and sparse matrices, particularly useful for Hessian matrices of large dimensions employed in the Newton method. The inherent parallel nature of the algorithm enables its parallel implementation on GPUs, which has by experiments on kernels demonstrated superior runtime efficiency compared to sequential CPU execution.

References

[1] Marcus J. Grote and Thomas Huckle. "Parallel Preconditioning with Sparse Approximate Inverses". In: *SIAM Journal on Scientific Computing* 18.3 (1997). DOI: 10.1137/S1064827594276552. URL: https://doi.org/10.1137/S1064827594276552.

[2] Alexander Kallischko. "Modified Sparse Approximate Inverses (MSPAI) for Parallel Preconditioning". PhD thesis. Technische Universität München, Zentrum Mathematik, 2008. URL: https://d-nb.info/988321149/34.

[3] Matous Sedlacek. *Sparse Approximate Inverses for Preconditioning, Smoothing, and Regularization*. 2012. URL: https://mediatum.ub.tum.de/doc/1107998/426923.pdf.

[4] K. van Geffen. "Sparse Approximate Inverse Methods". PhD thesis. University of Groningen, 2013. URL: https://fse.studenttheses.ub.rug.nl/11132/1/Koen_van_Geffen_2013_TWB.pdf.

[5] Michael McAleer. "Advances in Decision Sciences". PhD thesis. Asia University, Taiwan, 2019. URL: https://iads.site/wp-content/uploads/papers/2019/Applications-of-the-Newton-Raphson-Method-in-Decision-Sciences-and-Education.pdf.

[6] Henrik Granau Holm and Henrik Laurberg Pedersen. *Lineær Algebra i Datalogi*. London, Storbritannien: Pearson, 2020.

[7] Kasper Unn Weihe, Kristian Quirin Hansen, and Peter Kanstrup Larsen. "Linear Algebra in Futhark". PhD thesis. University of Copenhagen, 2021. URL: https://futhark-lang.org/student-projects/kristian-kasper-peter-project.pdf.

[8] Jiaquan Gao, Xinyue Chu, and Yizhou Wang. "HeuriSPAI: a heuristic sparse approximate inverse preconditioning algorithm on GPU". In: *CCF Transactions on High Performance Computing* (2023). DOI: 10.1007/s42514-023-00142-2. URL: https://doi.org/10.1007/s42514-023-00142-2.

[9] *CUDA C++ Programming Guide*. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[10] *CUDA cuBLAS*. URL: https://docs.nvidia.com/cuda/cublas/index.html.

[11] *diku-dk/howto/servers*. URL: https://github.com/diku-dk/howto/blob/main/servers.md.

[12] *Sparse matrices (scipy.sparse)*. URL: https://docs.scipy.org/doc/scipy/reference/sparse.html.

[13] Andrew Kerr, Dan Campbell, and Mark Richards. "QR decomposition on GPU". PhD thesis. Georgia Institute of Technology, Georgia Tech Research Institute, Unknown. URL: https://cpb-us-w2.wpmucdn.com/sites.gatech.edu/dist/5/462/files/2016/08/Kerr_Campbell_Richards_QRD_on_GPUs.pdf.