UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

MSc Thesis

Christian Bendix Fjordstrøm (wjx230)

Parallel Sparse Approximate Inverse in CUDA

Advisor: Cosmin Oancea

Handed in: May 31, 2026

Abstract

The parallel Sparse Approximate Inverse (SPAI) algorithm first presented by Grote and Huckle provides an algorithm suited for parallel hardware which computes a preconditioner for the solution of sparse linear systems of equations. This thesis presents a data-parallel CUDA implementation of the SPAI algorithm. We examine the theory behind the algorithm and how each step can be implemented efficiently on highly parallel hardware.

The SPAI preconditioner implementation is evaluated by comparing it with cuSPARSE's incomplete LU factorization preconditioner. The preconditioner construction time and solver time are compared across a set of sparse matrices. These benchmarks show that on some matrices, SPAI can reduce overall preconditioner construction time and solver time compared to cuSPARSE's ILU.

Contents

1 Introduction

Solving systems of sparse linear equations of the form

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

is a in many disciplines. If A is large and is represented as a dense matrix, then the time and space required to directly solve the system can be prohibitively expensive. A common alternative is to use iterative solvers instead. These algorithms start with an initial guess x_0 and iteratively improve it until reaching some stopping condition. The rate of convergence for iterative solvers depends on A, and for some matrices, iterative solvers might not converge. To alleviate this issue one can use a preconditioning matrix M and solve either the left or right preconditioned system

$$MAx = Mb, \quad \text{or } AMy = b,\ x = My$$

cuSPARSE [6] provides a preconditioner implementation for sparse systems that uses incomplete LU factorization. Applying an ILU preconditioner $M \approx LU$ requires triangular solves in each iterative solver iteration. This is not an operation that is easily parallelized and is thus not ideal in highly parallel hardware. Grote and Huckle [8] introduce an alternative preconditioner in which the approximate inverse of A, $M \approx A^{-1}$ is constructed. Applying this preconditioner requires computing sparse matrix vector multiplication, which exposes more parallelism, and thus is better for GPU execution. Rasmussen's BSc [11] further investigates how to efficiently implement sparse approximate inverse (SPAI) in CUDA. However, the implementation is not fully data-parallel, and is limited by memory usage and not fully implemented stopping criterion.

1.1 Contributions

This thesis builds on the work by Grote and Huckle [8] and Rasmussen [11] to develop a fully data-parallel implementation of the SPAI algorithm. In this thesis, we describe how to implement each step of the algorithm in CUDA, as well as how the QR factorization-related computations in SPAI can be performed without explicitly forming the Q by using the stored Householder reflectors instead [9]. Since Q is large, this reduces memory usage and reduces the number of memory accesses.

The SPAI implementation is evaluated against cuSPARSE ILU by computing the preconditioning matricies and using them as preconditioners for iterative solvers. In this thesis, the BICGStab iterative solver algorithm is used. The evaluation compares preconditioner total construction time and iterative solver time. It shows that for some matrices, using the SPAI preconditioner provides an overall preconditioner construction and solving time speedup.

1.2 Report Outline

Section 2, 3 and 4 explains the relevant theoretical background and the theoretical details of the SPAI algorithm. Section 5 explains each step of the CUDA implementation. Section 6 describes optimizations aimed at reducing preconditioner construction time. The CUDA implementation is evaluated against cuSPARSE ILU in section 7.

2 Background

2.1 Solving Systems of Linear Equations

2.1.1 Iterative Solvers

Given a system of linear equations of the form

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

an iterative solver starts with an initial guess x_0, and performs a series of iterations that update the guess x_i. x_i is accepted if $r_i = b - Ax_i$ satisfies

$$\frac{\|r_i\|_2}{\|r_0\|_2} < \text{tol}, \quad \text{tol} \in \mathbb{R}^+$$

2.1.2 Preconditioning

A preconditioner is a matrix M that is used to transform a system of linear equations such that the resulting system converges faster when solved with iterative solvers. There are two common forms of preconditioning, left preconditioning:

$$MAx = Mb$$

and right preconditioning:

$$AMy = b\,, \;\; x = My$$

If M is the inverse of A, then the system is already solved. If $M \approx A^{-1}$, then $AM \approx I$ and $MA \approx I$. This makes the system better conditioned, reducing overall solving time.

2.1.3 Applying Preconditioners in Iterative Solvers

A common approach to preconditioning is to use incomplete factorization, such as incomplete LU factorization ([8] p. 839). Incomplete LU factorization factors a matrix as the product of a lower triangular (L) and upper triangular (U) such that $LU \approx A$. The preconditioned system becomes $(LU)^{-1}Ax = (LU)^{-1}b$. Since $(LU)^{-1}$ is not explicitly formed, the preconditioner is applied by solving triangular systems of the form $z = (LU)^{-1}v$ in each step of the preconditioner. This is done using two triangular solves:

$$Ly = v, \qquad Uz = y$$

These operations contain data dependencies and therefore expose less parallelism, making them less suited to execution on massively parallel hardware. An alternative is to use approximate inverse preconditioners that explicitly form $M \approx A^{-1}$.

When explicitly forming $M \approx A{-}1$, iterative solvers need to compute $z = Mv$ in each step of the solver which can be done with sparse matrix vector multiplication. Since sparse matrix vector multiplication exposes more parallelism, this approach can result in solver speedups on highly parallel hardware.

Thus, the primary factors the determine overall solving speed are:

- Constructing the preconditioner M
- Computing $z = Mv$ in each solver iteration
- Preconditioner quality i.e. how much the preconditioner reduces the number of solver iterations

2.2 Sparse Matrix Storage

Compressed sparse row (CSR) and compressed sparse column (CSC) are ways to represent sparse matrices such that less memory is used compared to dense storage. Both store matrices using three one-dimensional arrays. For a sparse $m \times n$ matrix A, let nnz be the number of non-zeros in A. A matrix stored in CSR format stores the following:

- An array of length nnz that stores the values of A.
- An array of length nnz that stores the column indices of the values.
- An array of length $m + 1$ that stores the index in the value and column index array where a given row starts.

CSC is analogous to CSR. Instead of storing column indices and indices where rows start, it stores row indices and indices where columns start. CSR and CSC can be used to access the non-zero values of specific rows and columns respectively in a cache-friendly manner.

2.3 QR Factorization Using Householder Reflections

QR factorization decomposes $A = QR$ where Q is an orthogonal matrix and R is upper triangular. Householder reflections have the form

$$P_i = I - \frac{2}{v_i^H v_i} v_i v_i^H$$

where v is a Householder vector [9]. Q and R from the decomposition of A are then

$$Q = P_1^H P_2^H \cdots P_{n-1}^H$$
$$R = P_{n-1} \cdots P_2 P_1 A$$

Implementations such as cuSOLVER's geqrf [4] represent the Householder reflections by v_i and a scaling factor τ_i.

2.4 CUDA Programming Model

CUDA is a programming model which has been developed by NVIDIA that allows programmers to write programs that run on NVIDIA GPUs. CUDA is implemented as an extension to the C++ programming language, and allows the programmer to define special functions, known as kernels, that are executed on the GPU. When launching a kernel, the programmer defines the number of CUDA threads which should execute the function, and all threads execute the given kernel in parallel [3].

2.4.1 Thread Hierarchy

In CUDA, threads are divided into a number of blocks defined by **blockDim**, each block is assigned an index by **blockIdx** and each thread is assigned an index by **threadIdx**. Each of these is a 3-dimensional vector with fields **x**, **y** and **z** for easy mapping to different problem dimensions. For example, when using only the first dimension, a thread's unique identifier can be computed with **tid** $=$ **blockIdx.x** $\cdot$ **blockDim.x** $+$ **threadIdx.x**. The total number of threads is the product of the number of blocks and the number of threads [3].

Inside each block, threads are organized in warps of 32 threads that execute as single instruction multiple data, meaning that in each cycle, all threads in a warp execute the same instruction. Additionally, there are several functions that can be used to efficiently synchronize data and communicate between threads in a warp [3].

2.4.2 Memory Hierarchy

A key part of the CUDA programming model is the distinction between *host* and *device*. The host is the C++ program that is executed on the CPU which launches kernels that are executed on the device (GPU). Host and device maintain separate memory spaces, and CUDA provides an API for allocating device memory (**cudaMalloc**) and copying data between the host and the device (**cudaMemcpy**) that work similarly to their C counterparts.

Memory allocated by **cudaMalloc** is known as *global memory* and is visible by all blocks. Each block has access to a limited amount of *shared memory* which

is visible by all threads within a block. Finally, each individual thread has a limited amount of *register memory*.

2.4.3 Writing Efficient CUDA Programs

To maximize performance of CUDA programs, the following factors must be considered.

2.4.3.1 Maximize Utilization An efficient CUDA program should make use of the available parallelism on the GPU.

2.4.3.2 Maximize Memory Throughput Similar to programs that run on a CPU, memory access is the dominant cost, and thus all memory access must be carefully considered. Generally, the most important concept is that of *coalesced access* to global memory. A program has coalesced access if all threads in a warp access consecutive words in global memory in the same load/store instruction (cite slide 8). This makes it such that the load/store is optimized by the hardware into a single memory transaction. This improves memory throughput by around a factor of 8 (cite CUDA 8.3.2). Register memory is even faster.

Due to being on-chip, shared memory has much higher bandwidth and lower latency than global memory, and should therefore be used when possible.

2.4.3.3 Avoid Memory Thrashing Allocating and releasing memory should be avoided during program execution by allocating an appropriately sized chunk of memory beforehand.

2.4.4 CUDA Libraries

NVIDIA provides several libraries that contain highly optimized functions for many common operations. Relevant libraries for this thesis are listed below:

- CUB provides primitive parallel functions such as sort, scan, filter, reduce etc. There are functions that perform these operations at both the device and block level [1].
- cuSPARSE contains basic linear algebra functions for sparse matrices such as incomplete LU factorization, sparse matrix matrix multiplication and sparse matrix vector multiplication [6].
- cuBLAS implements BLAS (Basic Linear Algebra Subprograms) and is used to solve upper triangular systems [2].
- cuSOLVER is a library for matrix decompositions and linear system solvers for dense and sparse matrices. It is used to compute QR factorizations [4].

- cuSolverDx provides device functions that perform QR factorization and linear solver functions [5].

3 The SPAI Algorithm

The SPAI algorithm presented in this section is based on presented by Grote and Huckle [8] as well implementation details from Emil Vilandt Rasmussen's BSc project [11]. A key part of the algorithm is to perform QR factorization and perform updates to Q and R. Grote and Huckle's paper leave out details about how to perform these updates, but Rasmussen's bachelor's thesis expands upon these details. Rasmussen explains how to perform these updates when explicitly forming Q. Forming Q is expensive, as both the operation of forming Q is expensive, and it makes it so matrix matrix and matrix vector multiplications have to be performed. Avoiding explicit Q also reduces memory requirements.

3.1 SPAI Objective

Given an initial sparsity M, the goal of the SPAI algorithm is to approximate $M \approx A^{-1}$ by minimizing the Frobenius norm of $AM - I$. With e_k being the k'th column vector of the identity matrix and m_k being the k'th column vector of M, by the definition of the Frobenius norm we have

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

Since each part of the component of the summation is independent, the problem can be solved for every column in A in parallel by finding the m_k''s that minimize

$$\min_{m_k} \|Am_k - e_k\|_2 \,, \quad k = 1, \ldots, n \tag{2}$$

The algorithm is divided into three steps. The first step solves the least squares problem using the initial sparsity (typically chosen as the identity matrix). The second stage checks if m_k satisfies the stopping criterion $\|Am_k - e_k\|_2 < \epsilon$ for some ϵ. If the stopping criterion is satisfied, then $M(:, k) = m_k$. Afterwards, the indices in m_k that lead to the biggest reduction in $\|Am_k - e_k\|$ are determined. The third step updates m_k by solving the updated least squares problem.

3.2 Complete SPAI Algorithm

`SparseApproximateInverse`$(A \in, M, MaxIter, \epsilon, \tilde{n}_2)$

Require: $A, M \in \mathbb{R}^{n \times n}$, tolerance ϵ, maximum iterations $MaxIter$, maximum number of new indices per iteration $\tilde{n}_2$

for every column m_k of M **do**
  $\mathcal{J} \leftarrow \{j \mid m_k(j) \neq 0\}$
  $\mathcal{I} \leftarrow \{i \mid A(i, \mathcal{J}) \neq 0\}$
  $n_1 \leftarrow |\mathcal{I}|$
  $n_2 \leftarrow |\mathcal{J}|$
  $Q, R \leftarrow \mathrm{QR}(A(\mathcal{I}, \mathcal{J}))$
  $e_k \leftarrow \mathcal{I}(:, k)$
  $\hat{e}_k \leftarrow e_k(\mathcal{I})$
  $\hat{c}_k \leftarrow Q_{\text{init}}^T \hat{e}_k$
  Solve upper triangular system $R_{\text{init}} \hat{m}_k \leftarrow \hat{c}_k(: n_2)$ for $\hat{m}_k$
  $r(\mathcal{I}) \leftarrow A(\mathcal{I}, \mathcal{J}) \hat{m}_k$
  $r(k) \leftarrow r(k) - 1$
  $iter \leftarrow 0$
  while $\|r\|_2 > \epsilon$ and $iter < MaxIter$ **do**
    $\mathcal{L} \leftarrow \mathcal{I} \cup \{k\}$
    $\mathcal{J}_{\text{cand}} \leftarrow \left(\bigcup_{\ell \in \mathcal{L}} \{j \mid A(\ell, j) \neq 0\}\right)$
    for all $j \in \mathcal{J}_{\text{cand}}$ **do**
      $\rho_j^2 \leftarrow \|r\|_2^2 - \dfrac{(r^T A(:, j))^2}{\|A(:, j)\|_2^2}$
    end for
    $\bar{\rho}^2 \leftarrow \frac{1}{|\mathcal{J}_{\text{cand}}|} \sum_{j \in \mathcal{J}_{\text{cand}}} \rho_j^2$
    $\tilde{\mathcal{J}} \leftarrow$ at most $\tilde{n}_2$ indices $j \in \mathcal{J}_{\text{cand}}$ with smallest ρ_j^2 such that $\rho_j^2 < \bar{\rho}^2$
    $\tilde{\mathcal{I}} \leftarrow \{i \mid A(i, \mathcal{J}) \neq 0, i \notin \mathcal{I}\}$
    $\tilde{n}_1 \leftarrow |\tilde{\mathcal{I}}|$
    $A_u \leftarrow Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$
    $B_2 \leftarrow \begin{pmatrix} A_u(n_2 :, :) \\ A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix}$
    $Q_B, R_B \leftarrow \mathrm{QR}(B_2)$
    $Q \leftarrow \begin{pmatrix} Q & 0 \\ 0 & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{n_2} & 0 \\ 0 & Q_B \end{pmatrix}$
    $R \leftarrow \begin{pmatrix} R & A_u(: n_2, :) \\ 0 & R_B \end{pmatrix}$
    $\mathcal{I} \leftarrow \mathcal{I} :: \tilde{\mathcal{I}}$
    $n_1 \leftarrow n_1 + \tilde{n}_1$
    $\mathcal{J} \leftarrow \mathcal{J} :: \tilde{\mathcal{J}}$
    $n_2 \leftarrow n_2 + \tilde{n}_2$
    Solve upper triangular system $R \hat{m}_k \leftarrow \hat{c}_k(: n_2)$ for $\hat{m}_k$
    $r(\mathcal{I}) \leftarrow A(\mathcal{I}, \mathcal{J}) \hat{m}_k$
    $r(k) \leftarrow r(k) - 1$
    $iter \leftarrow iter + 1$
  end while
  $M(\mathcal{J}, k) \leftarrow \hat{m}_k$
end for

3.3 Initial Least Squares

This section explains lines 2-12 in algorithm 3.2.

Since A and m_k are sparse, the size of the least squares problem can be significantly reduced by filtering such that only non-zero rows and columns are kept. First, index sets that contain indices of non-zero rows and columns are defined. We define $\mathcal{J}$ to be the set of row-indices in m_k with non-zero values

$$\mathcal{J} = \{j \mid m_k(j) \neq 0\}$$

and $\mathcal{I}$ to be the set of row-indices in $A(:, \mathcal{J})$ with non-zero values

$$\mathcal{I}\{i \mid A(i, \mathcal{J}) \neq 0\}$$

Going forward, the following notation is used:

$$\begin{aligned}
\hat{m}_k &= m_k(J) \\
\hat{A} &= A(\mathcal{I}, \mathcal{J}) \\
\hat{e}_k &= e_k(I) \\
n_1 &= |\mathcal{I}| \\
n_2 &= |\mathcal{J}|
\end{aligned}$$

With this definition, solving equation 2 for m_k is equivalent to solving for $\hat{m}_k$:

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

This problem is solved using QR decomposition:

$$\hat{A} = QR$$

where $Q \in \mathbb{R}^{n_1 \times n_2}$ is lower triangular and $R \in \mathbb{R}^{n_2 \times n_2}$ is upper triangular. Defining

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

we have

$$\begin{aligned}
&\hat{A}\hat{m}_k - \hat{e}_k \\
&\Leftrightarrow \\
&QR\hat{m}_k - \hat{e}_k \\
&\Leftrightarrow \\
&R\hat{m}_k = Q^T \hat{e_k}(: n_2)
\end{aligned}$$

the solution is obtained by solving for $\hat{m}_k$:

$$R\hat{m}_k = \hat{c}_k(: n_2)$$

The residual is defined as:

$$r = A(:, \mathcal{J})\hat{m}_k - e_k$$

Since A only contains non-zeros in the row-indices in $\mathcal{I}$ and $\hat{A} = A(\mathcal{I}, \mathcal{J})$, we can instead compute the dense matrix vector product $\hat{A}\hat{m}_k$.

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

3.4 Sparsity Update

This section explains lines 15-23 in algorithm 3.2.

If $\|r\|_2 < \epsilon$, then we finish and scatter $\hat{m}_k$ to $M(\mathcal{J}, k)$. Otherwise, we continue updating m_k.

Let $\mathcal{L} = I \cup \{k\}$. For every $l \in L$ there exists a set $\mathcal{J}_l$ such that $\mathcal{J}_l$ contains all non-zero indices of $A(l, :)$. We define

$$\mathcal{J}_{cand} = \bigcup_{l \in \mathcal{L}} \mathcal{J}_l$$

$\mathcal{J}_{cand}$ contains the candidate indices that can be added to $\mathcal{J}$ to reduce $\|r\|_2$. The elements in $\mathcal{J}_{cand}$ are chosen such that $\|r\|_2$ is minimized. This is done by finding μ_j that minimizes

$$\min_{\mu_j} \|r + \mu_j A(:, j)\|_2$$

which is solved by

$$\mu_j = -\frac{r^T A(:, j)}{\|A(:, j)\|_2^2}$$

For each j, the norm of the new residual $r + \mu_j A(:, j)$ is

$$\rho_j^2 = \|r\|_2^2 - \frac{(r^T A(:, j))^2}{\|A(:, j)\|_2^2}$$

We define $\tilde{\mathcal{J}}$ be the set of a number of $\tilde{n_2}$ indices $j \in \mathcal{J}_{cand}$ with the lowest ρ^2 where

$$rho^2 \leq \frac{1}{|\mathcal{J}_{cand}|} \sum_{i=1}^{\mathcal{J}_{cand}} \rho_i^2$$

.

$\tilde{n_2}$ is chosen arbitrarily, but a good heuristic is 5 ([11], 2.2). The $\tilde{\mathcal{I}}$ defined in the algorithm at line 22 is then set of indices of non-zero elements in $A(:, \tilde{\mathcal{J}})$ that are not in I and $\tilde{n_1} = |\tilde{\mathcal{I}}|$.

3.5 QR Update

This section explains lines 24-35 in algorithm 3.2.

After determining which indices to add, we must find the $m'_k(J \cup \tilde{\mathcal{J}})$ that minimizes the updated least squares problem:

$$\min_{m'_k(J \cup \tilde{\mathcal{J}})} \|A(\mathcal{I} \cup \tilde{\mathcal{I}}, \mathcal{J} \cup \tilde{\mathcal{J}}) m_k(J \cup \tilde{\mathcal{J}} - e_k(\mathcal{I} \cup \tilde{\mathcal{I}})\|$$

$I \cup \tilde{\mathcal{I}}$ and $\mathcal{J} \cup \tilde{\mathcal{J}}$ denote the updated row and column sets. In order to avoid having to compute the full problem in every update iteration, the order of $\mathcal{I}$ relative to $\tilde{\mathcal{I}}$ and $\mathcal{J}$ relative to $\tilde{\mathcal{J}}$ must be preserved. To accomplish this, we define an ordered union operator :: that performs concatenation where duplicates are removed. Since the order of elements in $\mathcal{I}$ and $\mathcal{J}$ does not change the least squares problem, using :: instead of $\cup$ is valid.

At iteration i we therefore have the following updates

$$\mathcal{I}_i = \mathcal{I}_{i-1} :: \tilde{\mathcal{I}}_i$$
$$\mathcal{J}_i = \mathcal{J}_{i-1} :: \tilde{\mathcal{J}}_i$$

The goal is to find m'_k that minimizes $\|A(\mathcal{I} :: \tilde{\mathcal{I}}, \mathcal{J} :: \tilde{\mathcal{J}}) m'_k - e_k\|_2$. Since the problem has already been solved for $A(\mathcal{I}, \mathcal{J})$ we can construct matrices such the updated part has to be solved. Using the following notation:

$$\tilde{A} = A(\mathcal{I} :: \tilde{\mathcal{I}}, J :: \tilde{\mathcal{J}}) = \begin{pmatrix} A(\mathcal{I}, \mathcal{J}) & A(\mathcal{I}, \tilde{\mathcal{J}}) \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix}$$
$$\tilde{m}_k = m_k(J :: \tilde{\mathcal{J}})$$
$$\tilde{e}_k = e_k(I :: \tilde{\mathcal{I}})$$

and the known QR decomposition of $A(\mathcal{I}, \mathcal{J})$ we get

$$\tilde{A} = \begin{pmatrix} Q & \\ & \mathcal{I}_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} R & Q^T A(\mathcal{I}, \tilde{\mathcal{J}} \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}} \end{pmatrix}$$

Using $A_u = Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$ we then have

$$\begin{pmatrix} Q^T & \\ & \mathcal{I}_{\tilde{n}_1} \end{pmatrix} \tilde{A} = \begin{pmatrix} R & A_u(: n_2, :) \\ 0 & A_u(n_2 :, :) \\ 0 & A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}}) \end{pmatrix}$$

We denote

$$B_1 = \begin{pmatrix} A_u(: n_2, :) \end{pmatrix}$$
$$B_2 = \begin{pmatrix} A_u(n2 :, :) \\ A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}} \end{pmatrix}$$

so

$$\begin{pmatrix} Q^T & \\ & \mathcal{I}_{\tilde{n}_1} \end{pmatrix} \tilde{A} = \begin{pmatrix} R & B_1 \\ 0 & B_2 \end{pmatrix}$$

Since R is already upper triangular from the previous QR decomposition, we only need to make B_2 upper triangular to make the right-hand side upper triangular. This is done by decomposing $B_2 = Q_B R_B$ and multiplying both sides by $\begin{pmatrix} \mathcal{I}_{n_2} & \\ & Q_B^T \end{pmatrix}$ such that only B_2 is multiplied by Q_B^T. Since $R_B = Q_B^T B_2$, this gives

$$\begin{pmatrix} I_{n_2} & \\ & Q_B^T \end{pmatrix} \begin{pmatrix} Q^T & \\ & I_{\tilde{n}_1} \end{pmatrix} \tilde{A} = \begin{pmatrix} R & B_1 \\ 0 & R_B \end{pmatrix}$$

$$\iff$$

$$\tilde{A} = \underbrace{\begin{pmatrix} Q & \\ & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{n_2} & \\ & Q_B \end{pmatrix}}_{Q_{\text{new}}} \underbrace{\begin{pmatrix} R & B_1 \\ 0 & R_B \end{pmatrix}}_{R_{\text{new}}} \tag{3}$$

With Q_{new} obtained from equation 3 we further define

$$\tilde{c}_k = Q_{new}^T \tilde{e}_k$$

Using R_{new} obtained from equation 3 we can then solve Equation 4 for $\tilde{m}_k$.

$$R_{new} \tilde{m}_k = \tilde{c}_k(: n_2 + \tilde{n}_2) \tag{4}$$

The sparsity update and QR update steps are repeated until $\|r\|_2 < \epsilon$ or the maximum number of iterations is reached.

Regarding Q and Q_B, if explicitly forming Q, Q is updated as outlined in [11] section 2.4. If Q is not explicitly formed, Q can be used as explained in section 3.5.1.

3.5.1 Multiplying Q^T using Householder reflectors

At the beginning of update iteration i, the set $\mathcal{I}_i$ has not yet been created, so we have

$$\mathcal{I}_{i-1} = \mathcal{I}_0 :: \tilde{\mathcal{I}}_0 :: \tilde{\mathcal{I}}_1 :: \cdots :: \tilde{\mathcal{I}}_{i-1}$$

With Q_{i-1} representing the accumulated Q_B's from previous QR updates, the QR update for update iteration i requires computing

$$A_u = Q_{i-1}^T A(\mathcal{I}_{i-1}, \tilde{\mathcal{J}}_i)$$

Since forming Q explicitly is expensive, we want to store each iteration's Q_B as Householder reflectors.

Let $Q_B^{(j)}$ be the j'th update's orthogonal factor Q_B, and let $\bar{Q}_j$ be a matrix with $Q_B^{(j)}$ embedded such that it acts as the identity on the leading rows, and as $Q_B^{(j)}$ on the trailing rows.

$$\bar{Q}_j = \begin{pmatrix} I_{n_2^{(j)}} & 0 \\ 0 & (Q_B^{(j)}) \end{pmatrix}$$

With explicit Q, the following update would have been performed:

$$Q_{i-1} \leftarrow \begin{pmatrix} Q_{i-2} & 0 \\ 0 & I_{\tilde{n}_1} \end{pmatrix} \begin{pmatrix} I_{n_2} & 0 \\ 0 & Q_B^{(i-1)} \end{pmatrix}$$

An equivalent transformation as multiplication with Q_{i-1}^T can be realized by multiplying each $\bar{Q}_j$ in sequence:

$$Q_{i-1}^T = \bar{Q}_{i-1}^T \bar{Q}_{i-2}^T \cdots \bar{Q}_0^T$$

Instead of forming each $\bar{Q}_j$, the Householder representation of $(Q_B^{(j)})^T$ can be applied directly on the affected rows:

$$A_u(n_2^{(j)} :, :) \leftarrow (Q_B^{(j-1)})^T A(\mathcal{I}_{i-1}, \tilde{\mathcal{J}}_i)(n_2^{(j)} :, :)$$

Let Q_{init} being the result of QR decomposition of $\hat{A}$ in the Initial Least Squares step. In iteration i, A_u is computed as:

$$A_u \leftarrow Q_{\text{init}}^T A(\mathcal{I}_{i-1}, \tilde{\mathcal{J}}_i)$$

followed by

$$A_u(n_2^{(j)} :, :) \leftarrow (Q_B^{(j)})^T A(\mathcal{I}_{i-1}, \tilde{\mathcal{J}}_i)(n_2^{(j)} :, :), \qquad j \in \{k \in \mathbb{Z} \mid 0 \leq j < i\}.$$

Note that when $i = 0$, the set $\{k \in \mathbb{Z} \mid 0 \leq j < i\}$ is empty, so only Q_{init} is applied. $Q^T \hat{c}_k$ is computed in the same way, except $j \in \{k \in \mathbb{Z} \mid 0 \leq j \leq i\}$.

4 Implementation

This section describes how the SPAI algorithm is mapped to CUDA. It describes relevant design choices made for reducing runtime, as well as detailed explanations and snippets for relevant steps of the algorithm. This section first presents the loop transformations that are employed to make the algorithm better suited for parallel implementation. Afterwards, Section 4.2 explains the overall strategy for mapping the algorithm to CUDA. Sections 4.3 and 4.4 describe how device memory is allocated and which data structures are used. Finally, Sections 4.5-4.8 explain the implementation.

4.1 Loop Transformations

The algorithm in 3.2 is transformed to make it better suited for parallel execution. We note that the algorithm has the form

---

for every column m_k of M **do** ▷ par
  Statement 1
  ...
  while $\|r\|_2 > \epsilon$ and $iter < MaxIter$ **do** ▷ seq
    Statement 2
    ...
  end while
  Statement 3
end for

By Corollary 3 (Parallel Loop Distribution) ([10] 5.4), a parallel loop can be distributed across each one of its statements. This yields the following:

---

for every column m_k of M **do** ▷ par
  Statement 1
end for
...
for every column m_k of M **do** ▷ par
  while $\|r\|_2 > \epsilon$ and $iter < MaxIter$ **do** ▷ seq
    Statement 2
    ...
  end while
end for
for every column m_k of M **do** ▷ par
  Statement 3
end for

By Corollary 2 (Interchanging a Parallel Loop Inwards) ([10] 5.3), it is always safe to interchange a parallel loop inwards one step at a time. Since the for loop is parallel, it can be interchanged one step inwards. Similar to before, the for loop is also distributed across each statement:

---

for every column m_k of M **do** ▷ par
    Statement 1
end for
...
while $\|r\|_2 > \epsilon$ and $iter < MaxIter$ **do** ▷ seq
    for every column m_k of M **do** ▷ par
        Statement 2
    end for
    ...
end while
for every column m_k of M **do** ▷ par
    Statement 3
end for

By Theorem 9 (Loop Distribution) ([10] 5.4), array expansion must be performed for the variables that are used in multiple statements. How this is done is explained further in 4.3.

4.2 Parallelization Strategy

The loop distribution and loop interchange transformations change the algorithm such that each step is solved for all columns $m_k \in M$ in parallel. Thus, to maximize thread-level parallelism ([3] (8.2.3), the implementation uses the following strategies ranked by efficacy from high to low:

1. Using batched CUDA library functions. The CUDA libraries listed in Section 2.4.4 provide several highly efficient functions for computing batched operations. In the cases where batched CUDA library functions partially or fully solve the given problem, these functions are significantly more efficient than what could reasonably be implemented in a custom kernel.

2. Custom kernels where each block is mapped to a specific subproblem k i.e. $k =$ **blockIdx.x**. This strategy has both advantages and disadvantages. Since some subproblems will be larger than others, we risk having some subproblem not being assigned enough threads, and some subproblems being assigned too many threads. However, this strategy reduces warp divergence and makes it easier to guarantee coalesced access in padded-per-subproblem arrays since all threads in a warp are guaranteed to be assigned to the same subproblem. This also makes it possible to use faster shared memory in each subproblem, as well as CUB block-wide primitives. Additionally, due to the

simplicity of this strategy, it is the most commonly used in this implementation.

3. A sequential host-loop calling CUDA library functions. CUDA kernel launch is asynchronous, so a sequential host-loop can potentially utilize the GPU with asynchronous kernel launches if the problems are large enough.

4.3 Memory Allocation

To minimize memory thrashing from repeated device allocations, all workspaces are pre-allocated according to their maximum size. Let $nnz : \mathbb{R}^n \rightarrow \mathbb{N}$ be a function that returns the number of non-zero element in a vector. We observe the following about the size of $\mathcal{J}$ for a given k at iteration i where $i = 0$ means the Initial Least Squares step:

$$|\mathcal{J}_k^{(i)}| \leq nnz(m_k) \cdot i \cdot \tilde{n}_2$$

and therefore the maximum size for any $\mathcal{J}_k^{(i)}$ is

$$\mathcal{J}_{max}^{(i)} = \max_{0 \leq k < n} |\mathcal{J}_k^{(0)}| \cdot i \cdot \tilde{n}_2$$

From [11] we know that $\mathcal{I}_{max} = \mathcal{J}_{max} \cdot \max_{0 \leq k < n} nnz(A:, k)$. Therefore, for a given iteration i, we have:

$$\mathcal{I}_{max}^{(i)} = \mathcal{J}_{max}^{(i)} \cdot \max_{0 \leq k < n} nnz(A:, k)$$

When performing array expansion, we can therefore guarantee that all data will fit if the following sizes are allocated:

$$\begin{aligned}
&\mathcal{J} : n \times \mathcal{J}_{max}^{(MaxIter)} \\
&\mathcal{I} : n \times \mathcal{I}_{max}^{(MaxIter)} \\
&\hat{A} : n \times \mathcal{I}_{max}^{(0)} \times \mathcal{J}_{max}^{(0)} \\
&R : n \times \mathcal{J}_{max}^{(MaxIter)} \times \mathcal{J}_{max}^{(MaxIter)} \\
&\tilde{r}s : n \times \mathcal{I}_{max}^{(MaxIter)} \\
&A_u : n \times \mathcal{I}_{max}^{(MaxIter)} \times \tilde{n}_2 \\
&\hat{c} : n \times \mathcal{I}_{max}^{(MaxIter)} \\
&B_2^{(i)} : n \times ((\mathcal{I}_{max}^{(i)} - \mathcal{J}_{max}^{(i)}) + \max_{0 \leq k < n} nnz(A:, k)) \times \tilde{n}_2
\end{aligned}$$

4.4 Data Structures

4.4.1 Sparse Matrix Storage

The algorithm requires column-wise and row-wise access to A. Because of this, A is stored in both CSC and CSR format such that we can have coalesced access to A in both cases.

4.4.2 Segmented Index Sets

$\mathcal{I}s$ and $\mathcal{J}s$ are modeled using a data structure, henceforth referred to as a *segmented index set*. The segmented index set contains a shape descriptor that stores how many indices are in each segment's index set. We denote `max` as the maximum possible amount of indices as explained in Section 4.3. Each segment is allocated a block with space for `max` indices. The shape descriptor denotes how many are valid, and the rest are padded.

A segmented index set thus stores two arrays:

- `values`, an array of size $n \times$ `max` containing the stored indices;
- `shapes`, an array of size n storing the actual number of indices in each segment.

This is done to easily be able to index into each segments index sets. The shape array also stores the size of each index set in a convenient manner.

4.4.3 Dense Matrices and Vectors

For the sequence of matrices A_u, each matrix represented as a dense column-major matrix and sequence of vectors $\hat{r}$ and $\hat{c}$ are simply dense vectors. Similarly to segmented index sets, each segment is allocated a block of memory with size as described in Section 4.3. The contents of A_u, $\hat{r}$ and $\hat{c}$ are overwritten in each iteration, which means that every column can be stored next to each other in memory.

R is also represented as a dense column-major matrix, but to avoid writing the current contents of R in each update iteration, each column is stored in a padded format such that for the subproblem i, each column c starts at index $i \cdot \mathcal{J}_{max}^{(MaxIter)} \cdot \mathcal{J}_{max}^{(MaxIter)} + c \cdot \mathcal{J}_{max}^{(MaxIter)}$. This makes it so new columns can simply be added next to old columns, and new rows can simply be added after old rows.

4.4.4 QR Factors and Offsets

$\hat{A}$ and $B_2^{(i)}$ are stored as dense column-major matrices along with a τ vector. The values and shapes of $\mathcal{I}$, $\mathcal{J}$, $\tilde{\mathcal{I}}$ and $\tilde{\mathcal{J}}$ at the time the QR factorization of that iteration was computed are required to compute the size of $\hat{A}$ and $B_2^{(i)}$ as well as to compute

which offset Q_B should be applied to. Thus, in the initial least squares step, and in each update iteration, the matrix storing the QR factorization, the matrix storing τs, and vectors storing sizes and offsets are saved.

4.4.5 Segment Access

The data structures store arrays for each subproblem in a flat, padded-per-segment array. For such an array xs where each subproblem is allocated a block of size x_{stride}, the segment belonging to subproblem k starts at offset

$$xs_{offset} = k \cdot x_{stride}$$

Thus, the t'th element of segment k is stored at

$$xs[xs_{offset} + t]$$

In the CUDA implementation, this is typically implemented using pointer arithmetic:

```
auto x_offset = k * x_stride;
auto* x = xs + x_offset;
```

Then the t'th element is stored at `x[t]`. For readability, following notation is used in the following kernel descriptions:

$$x_k[t] = xs[k \cdot x_{stride} + t]$$

4.5 Initial Least Squares

4.5.1 Constructing $\mathcal{J}_k$

This step implements line 2 and 4 in Algorithm 3.2. For each subproblem, the indices of all non-zero rows in M are copied to $\mathcal{J}_k$. Since M is stored in CSC format, the row indices of non-zero elements in column k are stored in

$$[M.col\_ptrs[k], M.col\_ptrs[k+1])$$

This is implemented by the following kernel:

```
template <typename TValue, typename TIndex>
__global__ void batchedSetElementsToNonZeroIndices(
    SegmentedIndexSet<TIndex> Js,
    CSCMatrix<TValue> M)
{
    const auto k = blockIdx.x;
    const auto tid = threadIdx.x;

    const auto start = M.d_col_ptrs[k];
    const auto end = M.d_col_ptrs[k + 1];
    const auto nnz = end - start;

```

```
    if (tid < nnz)
        Js.values[k * Js.max + tid] = M.d_row_indices[start + tid
    ];

    if (tid == 0)
        Js.shapes[k] = nnz;
}
```

This access pattern ensures coalesced access for both reads from `M.row_indices` and writes to $\mathcal{J}s.values$. This is the case since within each warp of block k, consecutive threads read consecutive elements from $M.row\_indices$ and write to consecutive elements of $\mathcal{J}s.values$.

4.5.2 Constructing $\mathcal{I}_k$

This step implements line 3 and 5 in Algorithm 3.2. For each subproblem, the row indices of all non-zero elements in each of the columns in $\mathcal{J}_k$ are copied to $\mathcal{I}_k$. The primary issue is avoiding duplicates. Duplicates are eliminated with the following procedure:

- In parallel, each thread marks all row-indices that occur in the CSC row-index list of the columns in $\mathcal{J}_k$
- In parallel, all marked indices are written to $\mathcal{I}_k$

The CUDA implementation uses a shared-memory flag array with elements equal to rows in A. It is used to indicate which row-indices exist in $A(i, \mathcal{J}_k)$. This is done to easily eliminate duplicate elements while eliminating expensive non-coalesced global memory access. A warp is assigned to each $j \in \mathcal{J}_k$. Each block has a shared-memory counter variable `nextJ`. Each warp is assigned an index in $\mathcal{J}_k$ by having lane-zero increment `nextJ` using `atomicAdd` and broadcasting the value to the rest of the lanes using `__shfl_sync`.

```
int32_t j = 0;
if (lane == 0)
    j = atomicAdd(&nextJ, 1);

j = __shfl_sync(0xffffffffu, j, 0);

```

Each warp loops over the row-indices for the assigned column, and sets the flag for each row-index using `atomicExch`.

```
const auto col = Js.values[j_off + j];
for (auto p = start = A.d_col_ptrs[col] + lane; p < A.d_col_ptrs[
    col + 1]; p += WARP)
{
    const int32_t r = A.d_row_indices[p];
    atomicExch(&flag[r], 1u);
}
```

A block-wide exclusive scan of the flag array is computed using `cub::BlockScan::ExclusiveSum`, and the flag array is then used to scatter the exclusive scan result to $\mathcal{I}_k$. The shared memory variable, `runningTotalShared` is used to store how many elements are in the index set for index calculation and to store the shape:

```
\begin{lstlisting}
using BlockScan = cub::BlockScan<int32_t, 256>;
__shared__ typename BlockScan::TempStorage scanStorage;
__shared__ int32_t runningTotalShared;

if (tid == 0) runningTotalShared = 0;
    auto* I = Is.values + k * Is.max;

for (auto i = 0; i < A.n; i += blockDim.x)
{
    const auto t = i + tid;

    int32_t flagged = (t < A.n && mark[t]) ? 1 : 0;

    int32_t localOffset;
    int32_t count;

    BlockScan(scanStorage).ExclusiveSum(flag, localOffset, count);
    __syncthreads();

    if (flagged)
    {
        const auto pos = runningTotalShared + localOffset;
        I[pos] = t;
    }

    /* ... update runningTotalShared from count and set shape ...
    */
}
```

4.5.3 QR($A(\mathcal{I}_k, \mathcal{J}_k)$

This step implements line 6 in Algorithm 3.2. A is filtered using the indices in $\mathcal{I}$ and $\mathcal{J}$ and then the QR factorization is computed using cusolverDn<t>geqrf ([4] 2.4.2.8).

Each block is mapped to one subproblem k. For each column index $j \in \mathcal{J}_k$, all threads in the block iterate over the corresponding column in A and copy each element to $\hat{A}_k[j \cdot |\mathcal{I}_k| + i]$:

```
for (auto j = 0; j < Js.shapes[k]; ++j)
{
    const auto col = Js.values[k * Js.max + j];

    const auto start = A.d_col_ptrs[col];
    const auto end   = A.d_col_ptrs[col + 1];

```

```
    for (auto p = start + tid; p < end; p += blockDim.x)
    {
        const auto row = A.d_row_indices[p];
        const auto i = linearSearch(Is.values[k + Is.max + row],
    Is.shapes[k], row);

        A_hat[j * Is.max + i] = A.d_values[p];
    }
}

```

Since $\mathcal{I}$ is not sorted, linear search is used to find the index in $\mathcal{I}$ where the row-index is stored.

QR factorization is then performed on the filtered matrix. Finally, each R_k needs to be copied from the lower triangular part of $\hat{A}$ to storage in Rs:

```
for (int32_t k = 0; k < qrState.cols; ++k)
{
    TValue* A = qrState.d_A0s + k * qrState.A0Stride;
    TValue* tau = qrState.d_Tau0s + k * qrState.Tau0Stride;

    CUSOLVER_CHECK(cusolverDnXgeqrf(
        data.handle, data.params,
        m, n,
        dataType, A, m,
        dataType, tau,
        dataType, data.d_work, data.wsDev,
        data.h_work, data.wsHost,
        d_info + k
    ));
}
```

4.5.4 Triangular Solve

This step implements line 7-10 in Algorithm 3.2. It is implemented by computing $\hat{c}_k = Q_{init}^T \hat{e}_k$ where Q_{init} is stored as Householder reflectors. Afterwards, the system on line 10 is solved.

cusolverDn<t>ormqr ([4] 2.4.2.11) can be used to compute $C = Q^T \cdot C$ where Q is the result of `geqrf`. Since the matrix with which Q^T is multiplied is overwritten, $\hat{c}_k$ is first set to $\hat{e}_k$, and then `ormqr` is used to compute the product with $m = |\mathcal{I}_k|$,, $k = 1$ and $n = |\mathcal{J}_k|$. cusolverDn<t>geqrf stores the results of the QR factorization in place in $\hat{A}$ so in the following snippets both Q and R are stored in `qrState.d_A0`.

```
for (int32_t k = 0; k < cols; ++k)
{
    ...
    TValue* CHat = CHats + k * Is.max;
    TValue* A = qrState.d_A0s + k * qrState.A0Stride;
```

```
    TValue* tau = qrState.d_Tau0s + k * qrState.Tau0Stride;

    CUSOLVER_CHECK(cusolverDnDormqr(
        data.handle, CUBLAS_SIDE_LEFT, CUBLAS_OP_T,
        m, 1, n,
        A, m,
        tau,
        CHat, m,
        (double*) data.d_work, data.wsDev,
        d_info + k
    ));
}
```

Afterwards $R_k \hat{m} = \hat{c}_k$ is solved for $\hat{m}_k$ using cublas<t>trsv with $n = |\mathcal{I}_k|$. cublas<t>trsv overwrites the right-hand-side, so $\hat{c}$ is used to store $\hat{m}$:

```
for (int32_t k = 0; k < n_cols; ++k)
{
    ...
    int32_t m = qrState.h_m0s[k];
    int32_t n = qrState.h_n0s[k];

    TValue* A = qrState.d_A0s + k * qrState.A0Stride;
    TValue* CHat = CHats + k * Is.max;

    CUBLAS_CHECK(cublasDtrsv(
        data.handle,
        CUBLAS_FILL_MODE_UPPER, CUBLAS_OP_N, CUBLAS_DIAG_NON_UNIT,
        n,
        A, m,
        CHat, 1
    ));
}
```

4.5.5 Computing the Residual

The residual is computed as a matrix vector multiplication.

4.6 Sparsity Updates

Subproblems are solved when $\|r_k\| \leq \epsilon$. Since some subproblems are solved earlier than others, we must be able to keep track of which subproblems are unsolved. This is solved by maintaining a variable `NumActive` which stores the number of unsolved subproblems, and an array `active` which stores active subproblem numbers $k \in \{0, \ldots, n-1\}$ in the first `NumActive` elements of the array. The active subproblems can updated with a filter operation that keeps subproblems that do not satisfy the stopping criteria in algorithm 3.2 line 14, i.e. subproblems where $\|r_k\|_2 > \epsilon$. This is implemented using `cub::DeviceSelect::FlaggedIf` [1]:

```
cub::DeviceSelect::FlaggedIf(
```

```
        d_activeTempStorage, activeSetTempBytes,
        activeIn, activeNorms, activeOut,
        d_numActive, h_numActive,
        GreaterThanEpsilonOp<T>(epsilon));
```

4.6.1 Constructing $\mathcal{J}_{cand}$

$\mathcal{J}_{cand}$ is constructed in the same way as $\mathcal{I}$, except that the CSR representation of A is used and the column indices in the rows in $\mathcal{I}_k \cup k$ are considered.

4.6.2 Constructing $\tilde{\mathcal{J}}$

For each $j \in \mathcal{J}_{cand}$, this step computes ρ_j^2 as shown in algorithm 3.2 lines 17-19, and then selects the at most $\tilde{n}_2$ j's with smallest ρ_j^2 where ρ_j^2 is less than the average ρ^2, corresponding to lines 20-21 in algorithm 3.2. It is constructed by the following steps:

- Each thread gets a set of j's from $\mathcal{J}_{cand}$
- Each thread computes ρ_j^2 in register memory
- Each thread computes the sum of its ρ_j^2's
- Each block computes the average ρ_j^2 using BlockReduce::Sum and having thread 0 divide by $|\mathcal{J}_{cand}|$
- Sort j's by ρ_j^2 using BlockRadixSort::SortBlockedToStriped
- thread 0 writes lowest 5 to $\tilde{\mathcal{J}}$

4.6.3 Constructing $\tilde{\mathcal{I}}$

This step implements line 22 in algorithm 3.2. The implementation is the same as that for $\mathcal{I}$ (line 3) with the exception that elements in $\mathcal{I}$ should not be added to $\tilde{\mathcal{I}}$.

4.7 QR Updates

4.7.1 Computing A_u

To compute $Q^T A(\mathcal{I}, \tilde{\mathcal{J}})$ where Q^T is stored as Householder reflectors, we apply the stored Householder reflectors in the order they were generated. Let Q_{init} be Householder reflectors from the Least Initial Squares step obtained from $QR(A(\mathcal{I}, \mathcal{J}))$ in algorithm 3.2 line 6. Let Q_i be the Householder reflectors from iteration i of the QR Update obtained from $QR(B_2)$ in Algorithm 3.2 line 26. Each Q_i^T must be multiplied with $A(\mathcal{I}, \tilde{\mathcal{J}})$ at row $n_2^{(i)}$.

`ormqr` is used to compute the multiplications. Let C be a pointer to a matrix where ldc is the leading dimension of the two-dimensional array used to store C. For each column index j in C, the first element of the column is addressed as $C[j * ldc]$. Thus, multiplying reflectors with the submatrix starting at row r in C is done by passing a pointer to $C + r$. This makes it so the first element of column j is addressed as $C[r + j * ldc]$.

For simplicity, we define

$$\text{ormqr}(C, ldc, Q, m, n, k)$$

for multiplying k Householder reflectors stored in Q with an $m \times n$ submatrix C stored with leading dimension ldc. ormqr stores the results in C.

With A_u storing $A(\mathcal{I}, \tilde{\mathcal{J}})$, and $ldA_u = |\mathcal{I}|$ the multiplication then becomes:

`ormqr(` $A_u, ldA_u, Q_{init}, n_1^{(init)}, \tilde{n}_2^{(i)}, n_2^{(init)}$)
for $it = 0..i-1$ **do**
    `ormqr(` $A_u + n_2^{(it)}, ldA_u, Q_{it}, n_1^{(it)} - n_2^{(it)} + \tilde{n}_1^{(it)}, \tilde{n}_2^{(i)}, \tilde{n}_2^{(it)}$)
end for

4.7.2 Constructing B_2

In this step B_2 is constructed as in algorithm 3.2 line 25. $A_u(n_2 :, :)$ is copied into B_2 and the result of the filter operation $A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ is written directly into B_2.

A_u has n_1 rows. Since the rows starting at row n_2 are copied, $n_1 - n_2$ rows in total need to be copied. Since A_u is stored as column-major, and we only want to copy certain rows, there is an offset in memory between each column. Thus, to ensure coalesced access, we use a warp of threads to copy an entire column so different warps copy different columns.

```
const auto lane = threadIdx.x & (31);
const auto warpId = threadIdx.x >> 32;

for (auto row = lane; row < (n1 - n2); row += 32)
{
    const auto AuRow = row + n2;
    B2[warpId * ((n1 - n2) + n1tilde) + row] = Au[warpId * n1 +
    AuRow];
}
```

$A(\tilde{\mathcal{I}}, \tilde{\mathcal{J}})$ is constructed in the same way as previous filter operations.

4.7.3 QR(B_2)

The QR factorization in line 26 of algorithm 3.2 is computed in the same way as the QR factorization in the Initial Least Squares step, and the result is saved such

that future iterations can access it.

4.7.4 Updating R

The copies needed to update R in line 28 in algorithm 3.2 are similar to the warp-per-column copy in B_2. Since R has size $n_2 \times n_2$ the copies should copy into R with column offset n_2. $A_u(: n_2)$ is copied with a warp per column:

```
const auto lane = threadIdx.x & (31);
const auto warpId = threadIdx.x >> 5;

for (auto row = lane; row < n2; row += WARP)
{
    R[(n2 + warpId) * Js.max + row] = Au[warpId * n1 + row];
}
```

R_B is also copied with a warp per column. Since R_B is upper triangular, only elements where the row index is less than or equal to the column index are copied:

```
const auto lane = threadIdx.x & (31);
const auto warpId = threadIdx.x >> 5;

for (auto row = lane; row < n2Tilde; row += WARP)
{
    if (row <= warpId)
        RR[(n2 + warpId) * Js.max + (n2 + row)] = QRB[warpId * ((
    n1 - n2) + n1Tilde) + row];
}
```

4.7.5 Updating $\mathcal{I}$ and $\mathcal{J}$

This step computes lines 29-32 in algorithm 3.2. Since $\mathcal{I} \cup \tilde{I}I \; =$ and $\mathcal{J} \cup \tilde{J}J \; =$, this is simply a concatenation operation. Starting at the offset SegmentedIndexSet.shapes[k], each thread copies from $\tilde{\mathcal{I}}$ to $\mathcal{I}$ then from $\tilde{\mathcal{J}}$ to $\mathcal{J}$. n_1 and n_2 are then set to $n_1 + \tilde{n}_1$ and $n_2 + \tilde{n}_2$ respectively.

```
template <typename TIndex>
__global__ void MergeSegmentedIndexSets(
    SegmentedIndexSet<int32_t> Is,
    SegmentedIndexSet<int32_t> Js,
    SegmentedIndexSet<int32_t> I_tildes,
    SegmentedIndexSet<int32_t> J_tildes,
    TIndex* active)
{
    auto k = active[blockIdx.x];

    if (threadIdx.x < I_tildes.shapes[k]) {
        Is.values[k * Is.max + Is.shapes[k] + threadIdx.x] =
            I_tildes.values[k * I_tildes.max + threadIdx.x];
    }

    if (threadIdx.x < J_tildes.shapes[k]) {
```

```
        Js.values[k * Js.max + Js.shapes[k] + threadIdx.x] =
            J_tildes.values[k * J_tildes.max + threadIdx.x];
    }

    if (threadIdx.x == 0) {
        Is.shapes[k] += I_tildes.shapes[k];
        Js.shapes[k] += J_tildes.shapes[k];
    }
}
```

4.7.6 Triangular Solve

The method for computing the triangular solve in the QR update step is identical to the method explained for Initial Least Squares.

4.7.7 Computing the Residual

The method for computing the residual in the QR update step is identical to the method explained for Initial Least Squares.

4.8 Assembling the Approximate Inverse

The approximate inverse M is assembled in CSC format since $\tilde{m}$ stores the approximate column vectors. From Rasmussen's Bachelor's Thesis [11] we know that M's columns pointers can be computed with an exclusive scan of $\mathcal{J}$'s shape descriptor. This is computed using `cub::DeviceScan::ExclusiveSum`:

```
CUDA_CHECK(cub::DeviceScan::ExclusiveSum(
    d_scan_temp,
    scan_temp_bytes,
    Js.shapes,
    M.d_col_ptrs,
    Js.n));
```

Since each $\mathcal{J}_k$ is not sorted, we need to perform a segmented key-value sort of $\mathcal{J}$ and $\tilde{m}_k$ which can be done with `cub::DeviceSegmentedRadixSort::SortPairs`. Since each $\mathcal{J}_k$ and $\hat{m}_k$ are stored in a padded-per-subproblem representation, all elements must be packed such that they are contiguous such that the data fits the `cub::DeviceSegmentedRadixSort::SortPairs` API. M's column pointers store which index each segments index set starts at. These are the exact offsets needed to store each $\mathcal{J}_k$ and $\hat{m}_k$ contiguously as shown in the following listing:

```
template <typename TValue, typename TIndex>
__global__ void PackSolutionToFlat(
    const SegmentedIndexSet<TIndex> Is,
    const SegmentedIndexSet<TIndex> Js,
    const TValue* __restrict__ MHats,
    const TIndex* __restrict__ offsets,
    TIndex* __restrict__ keys,
```

```
    TValue* __restrict__ vals)
{
    const TIndex k = blockIdx.x;
    const TIndex count = Js.shapes[k];

    const TIndex* Jk = Js.values + k * Js.max;
    const TValue* Mk = MHats + k * Is.max;
    const TIndex out0 = offsets[k];

    for (TIndex i = threadIdx.x; i < count; i += blockDim.x)
    {
        keys[out0 + i] = Jk[i];
        vals[out0 + i] = Mk[i];
    }
}
```

With $\mathcal{J}_k$ and $\tilde{m}_k$ being stored contiguously in the arrays `flatKeys` and `flatVals`, row indices and values of M are computed with `cub::DeviceSegmentedRadixSort::SortPairs`:

```
cub::DoubleBuffer<TIndex> keys(flatKeys, M.d_row_indices);
cub::DoubleBuffer<TValue> vals(flatVals, M.d_values);

CUDA_CHECK(cub::DeviceSegmentedRadixSort::SortPairs(
    d_sort_temp,
    sort_temp_bytes,
    keys,
    vals,
    nnz,
    Js.n,
    M.d_col_ptrs,
    M.d_col_ptrs + 1
));
```

5 Optimizations

5.1 cuSolverDx

The first implementation of qr-related (ormqr, geqrf and trsv) functions consists of a host-loop calling the relevant CUDA library function. Since B_2 has shape $n_1 + n_2 - \tilde{n}_1 \times \tilde{n}_2$ where $\tilde{n}_2 \leq 5$, many of the operations are small, leading to poor hardware utilization. cuSolverDx [5] provides device functions that perform QR factorization and linear solver functions. These functions can perform the operations cooperatively with all threads in a block. This is ideal when assigning each block one subproblem.

```
using Solver = decltype(
    cusolverdx::Size<M, N>() +
    cusolverdx::Precision<T>() +
    cusolverdx::Type<cusolverdx::type::real>() +
    cusolverdx::Arrangement<cusolverdx::col_major>() +
    cusolverdx::Function<cusolverdx::function::geqrf>() +
    cusolverdx::SM<SM>() +
    cusolverdx::BlockDim<BlockSize>() +
    cusolverdx::Block()
);
```

As can be seen in the listing, the size of the matrix that `geqrf` is performed on is part of the function template. Since this is a compile-time constant, the matrix sizes must be known at compile time, which they are not. This is solved by generating all required functions at compile-time and storing an array of device function pointers that can looked up at runtime. Since n_1, n_2, $\tilde{n}_1$ and $\tilde{n}_2$ depend on the matrix for which the approximate inverse is computed, one has to ensure that all required functions are generated.

6 Performance Evaluation

6.1 Correctness

The correctness of the CUDA implementation is validated by comparing the results of each step of the CUDA implementation with a sequential implementation. Additionally, the Frobenius norm matches what is expected in [8] and [11].

6.2 Experimental Setup

Performance is evaluated by solving systems of linear equations using BiCGStab. The following metrics are measured using the SPAI preconditioner described in this thesis and cuSPARSE's Incomplete LU preconditioner:

- Time to construct the preconditioner
- Time to solve system
- Number of solver iterations

The following matrices from SuiteSparse Matrix Collection are used:

- sherman1: $n = 1,000$, $nz = 3,750$
- sherman2: $n = 1,080$, $nz = 23,094$
- sherman3: $n = 5,005$, $nz = 20,033$
- sherman4: $n = 1,104$, $nz = 3,785$
- sherman5: $n = 3,312$, $nz = 20,793$
- pores_2: $n = 532$, $nz = 3,474$
- pores_3: $n = 1,224$, $nz = 9,613$
- orsirr_1: $n = 1,030$, $nz = 6,858$
- orsirr_2: $n = 886$, $nz = 5,970$
- saylr3: $n = 1,000$, $nz = 3,750$
- saylr4: $n = 3,564$, $nz = 22,316$

For shermanx, the right-hand-side was given, for the rest, the right-hand-side is a random vector.

All experiments were performed using double precision floating point numbers. The initial guess was always $x_0 = \mathbf{0}$ with BiCGStab stopping criterion

$$\frac{\|b - Ax_i\|_2}{\|b\|} < 10^{-8}$$

except for pores_2 and sherman2 where a tolerance of 10^{-5} was used due to BiCGStab not converging.

All experiments were performed on an NVIDIA A100-PCIE-40GB GPU with 1.55GB/s memory bandwidth and 9.7 TFLOPS for FP64.

6.3 Experimental Results

6.3.1 SPAI Preconditioner

6.3.1.1 CuSolverDx Optimization The performance impact of the CuSolverDx optimization is measured by comparing preconditioner computation time with and without the optimization. For all matrices $\epsilon = 0.01$ and $MaxIter = 10$.

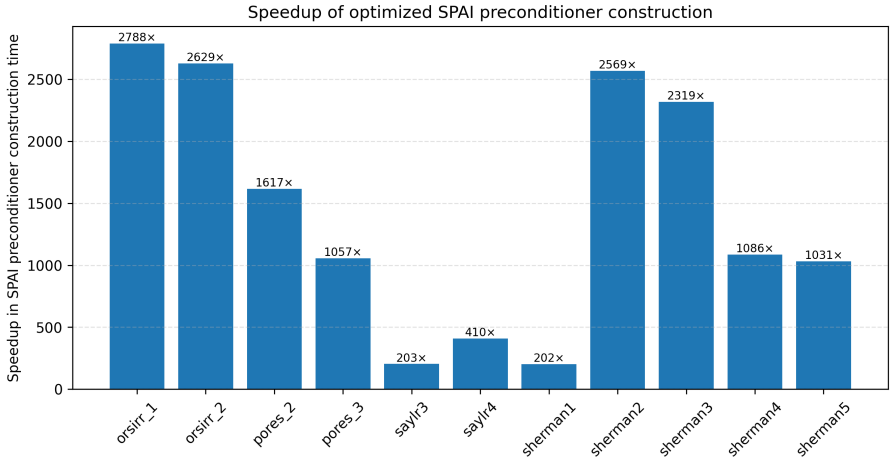

Figure 1: Speedup when using the CuSolverDx compared to the initial host-loop implementation for all test matrices using $\epsilon = 0.01$ and $MaxIter = 10$

As can be seen, due to the initial implementation being mostly sequential, the optimization yields a speedup in preconditioner computation time of around 200-3000x depending on the matrix.

6.3.1.2 SPAI Parameter Choice As can be seen from figure 2, as ϵ gets lower and as the the number of SPAI iterations increases, the number of solver iterations decreases. This is as expected, since the lower ϵ is and the more SPAI iterations are computed, the closer the computed preconditioner is to the inverse.

As can be seen from figure 3, lower solver iteration count does not necessarily mean faster total solving time, as too many SPAI iterations makes preconditioner computation becomes more expensive. Thus, the best preconditioner is not always the fastest.

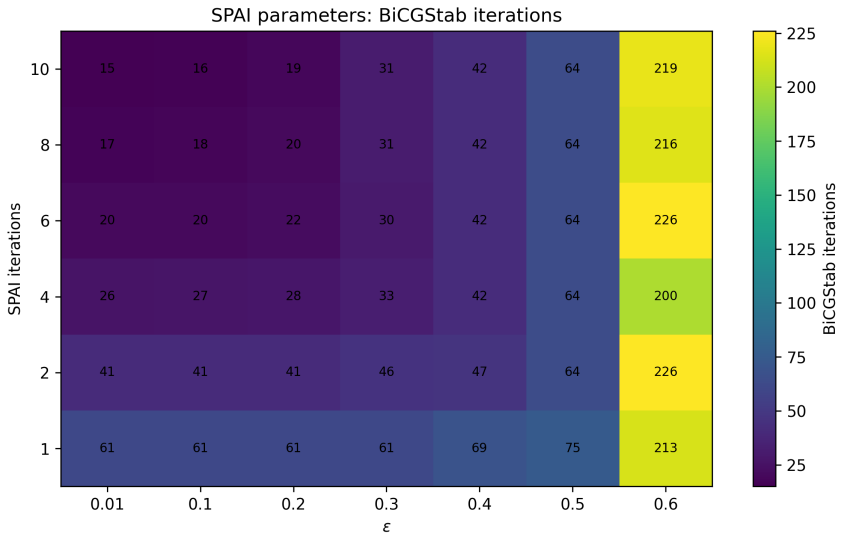

Figure 2: orsirr_1: BiCGStab solver iterations with SPAI preconditioner as a function of ϵ and SPAI iterations

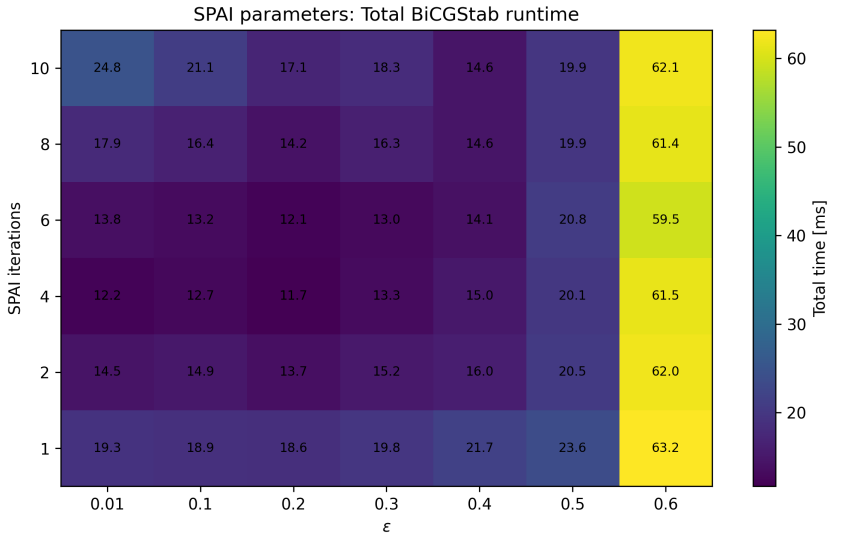

Figure 3: orsirr_1: BiCGStab total time to construct SPAI preconditioner and solve system as a function of ϵ and SPAI iterations

6.3.2 SPAI vs. cuSPARSE Preconditioner

In the following graphs, for each matrix, the best performing parameter setup for those matrices were selected for SPAI.

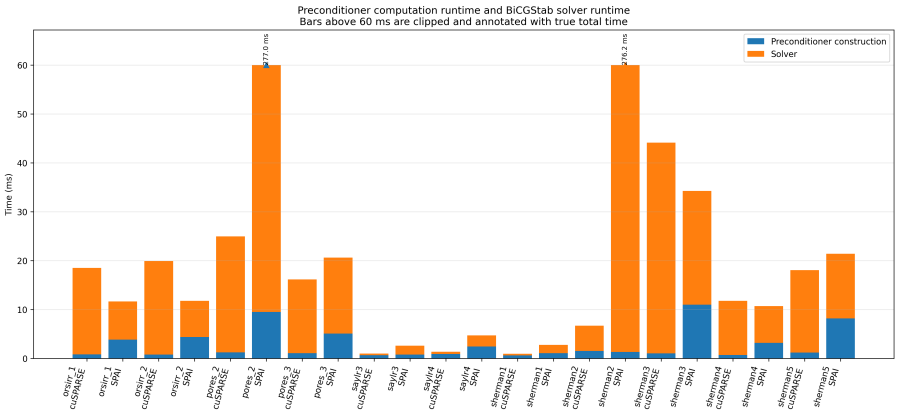

Figure 4: Preconditioner construction time and BiCGStab solver time using cuSPARSE preconditioner and SPAI preconditioner on different matrices.

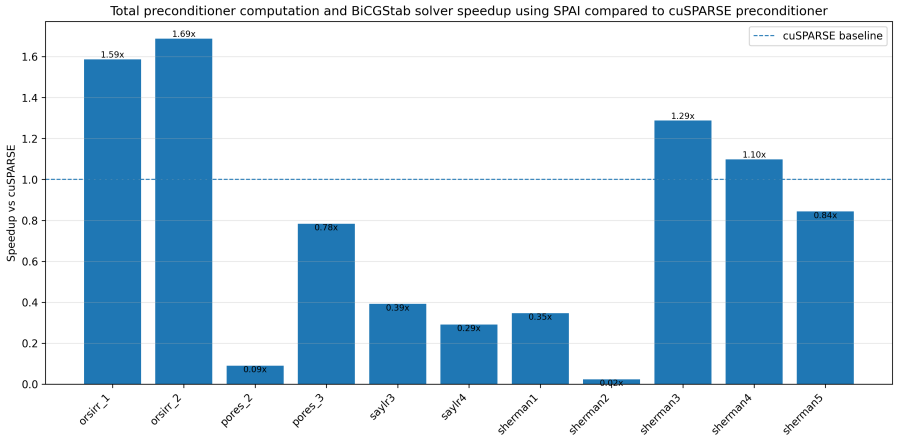

Figure 5: Speedup in total preconditioner construction and BiCGStab solver runtime. For readability,

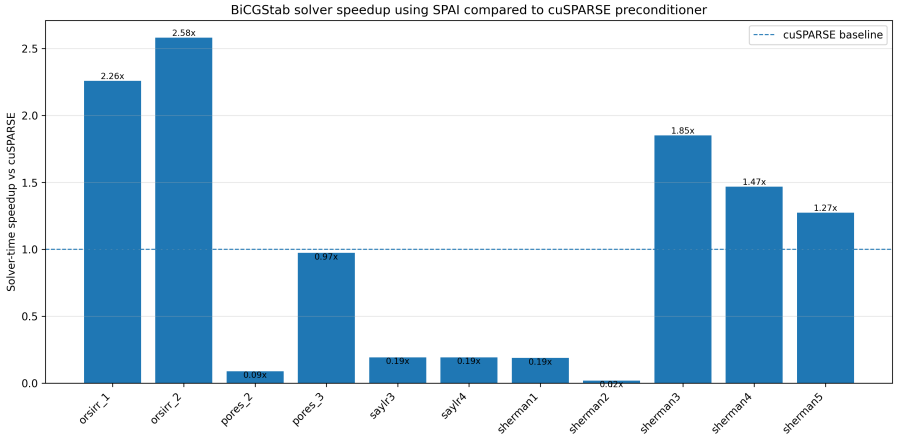

Figure 6: Speedup in total BiCGStab solver runtime

Grote and Huckle's results [8] show that solving sherman2 and pores_2 does not converge when using BiCGStab with a tolerance of 10^{-8}. This observation is also made here.

As can be seen in figure 4 and figure 6, using the SPAI preconditioner can result in an overall speedup for some matrices over cuSPARSE ILU. However, for other matrices, BiCGStab with SPAI preconditioner does not converge while ILU does which makes using the SPAI preconditioner slower than cuSPARSE ILU. Furthermore, for some matrices, cuSPARSE ILU still provides a significant speedup.

Matrix	ϵ	MaxIter	SPAI Memory Footprint (GB)	nnz(M)/nnz(A)	$\|AM - I\|_F$	SPAI (ms)	cuSPARSE (ms)	Speedup
orsirr_1	0.20	4	0.98	1.87	7.040	11.676	18.524	1.587×
orsirr_2	0.10	4	0.96	2.48	6.344	11.810	19.925	1.687×
pores_2	0.30	8	1.51	2.34	11.758	277.042	24.974	0.090×
pores_3	0.10	6	0.95	4.33	5.107	20.621	16.163	0.784×
saylr3	0.20	1	0.90	0.75	5.259	2.615	1.025	0.392×
saylr4	0.10	1	1.10	0.55	13.101	4.736	1.378	0.291×
sherman1	0.30	2	0.91	0.81	3.773	2.768	0.960	0.347×
sherman2	0.01	1	0.93	0.26	24.533	276.201	6.723	0.024×
sherman3	0.20	4	1.48	2.33	10.200	34.292	44.141	1.287×
sherman4	0.20	4	0.95	2.41	4.356	10.737	11.786	1.098×
sherman5	0.20	6	1.64	0.96	7.533	21.427	18.072	0.843×

Table 1: Summary of SPAI and cuSPARSE preconditioned BiCGStab. For SPAI, the ϵ and MaxIter that resulted in the lowest overall preconditioner construction and iterative solver time were selected. Memory usage, nnz(M)/nnz(A), and Frobenius norm are reported for the SPAI preconditioner. SPAI (ms) and cuSPARSE (ms) show overall preconditioner construction and iterative solver time

7 Discussion and Further Work

7.1 Results

The experiments show that SPAI generally trades higher preconditioner computation time for lower solving time compared to cuSPARSE ILU. Therefore, the combined runtime should be considered, and not just preconditioner time or iterative solver time in isolation.

SPAI performance is also heavily dependent on the matrix itself and on SPAI parameters i.e. maximum SPAI update iterations and ϵ. Higher ϵ and maximum SPAI iterations improves preconditioner quality, but makes preconditioner construction too expensive for it to be worth it for a single BiCGStab solve. If the same matrix is solved for multiple right-hand-sides, then preconditioner construction time can be amortized across many solves. In this case SPAI, especially with lower ϵ and higher maximum iterations, is an attractive choice.

SPAI can also solve problems that cuSPARSE ILU cannot. In the context of automatic differentiation, SPAI can be used to compute the inverse Hessian, which ILU cannot.

7.2 Limitations

Due to the way `geqrf`, `trsv` and `ormqr` are implemented, the matrix size on which this implementation can be used is limited. This is due to the optimization having to generate a function pointer for every required size at compile-time. The current shared memory usage in constructing $\mathcal{I}$ and $\tilde{\mathcal{I}}$ also restricts matrix size.

7.3 Future Work

Some of the dimensions in `geqrf`, `trsv` and `ormqr` are limited to 5 due to $|\tilde{\mathcal{J}}| \leq 5$. A different implementation of these functions could eliminate the compile-time function generation and make use of some of the dimensions being fixed to create an efficient implementation. To make the algorithm work on arbitrarily sized matrices, construction of $\mathcal{I}$ could instead be done by concatenating row-indices in the columns of $\mathcal{J}$ and then making the list unique, and simiarly for $\tilde{\mathcal{I}}$.

This thesis only compares iterative solver performance for BiCGStab where as Grote and Huckle compare iterative solver iteration count for GMRES, BCG, CG and BiCGStab.

Due to the previously mentioned matrix size restrictions, only matrices of size no larger than 5005×5005 have been benchmarked. Comparing SPAI with cuSPARSE on very large matrices such as the ones mentioned NVIDIA's description of ILU preconditioned iterative solvers [7] could be interesting.

8 Conclusion

In this thesis, we have presented the SPAI algorithm, and how to efficiently implement it in CUDA. This implementation improves on previous work by providing a fully data-parallel implementation with a lower memory footprint. Experimental results show, that for solving sparse systems of linear equations, using the SPAI preconditioner provides an overall preconditioner construction and BiCGStab solver time speedup over cuSPARSE ILU for several matrices. In particular, BiCGStab solving time is sped up when using the SPAI preconditioner.

References

[1] NVIDIA Corporation. CUB. https://nvidia.github.io/cccl/unstable/cub/index.html. Accessed: 2026-04-30.

[2] NVIDIA Corporation. cuBLAS. https://docs.nvidia.com/cuda/cublas/index.html. Accessed: 2026-04-30.

[3] NVIDIA Corporation. CUDA Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide. Accessed: 2026-04-30.

[4] NVIDIA Corporation. cuSOLVER. https://docs.nvidia.com/cuda/cusolver/index.html. Accessed: 2026-04-30.

[5] NVIDIA Corporation. cuSolverDx. https://docs.nvidia.com/cuda/cusolverdx/index.html. Accessed: 2026-04-30.

[6] NVIDIA Corporation. cuSPARSE. https://docs.nvidia.com/cuda/cusparse/index.html. Accessed: 2026-04-30.

[7] NVIDIA Corporation. Incomplete-lu and cholesky preconditioned iterative methods. https://docs.nvidia.com/cuda/pdf/Incomplete_LU_Cholesky.pdf. Accessed: 2026-05-04.

[8] Marcus J. Grote and Thomas Huckle. Parallel preconditioning with sparse approximate inverses. *SIAM Journal on Scientific Computing*, 18(3):838–853, 1996.

[9] Andrew Kerr, Dan Campbell, and Mark Richards. Qr decomposition on gpus. pages 71–78, 03 2009.

[10] Cosmin E. Oancea. Lecture notes for the software track of the pmph course. 2018.

[11] Emil Rasmussen. Sparse approximate inverse - a massively parallel implementation, 2023. Accessed: 2025-12-29.