INAUGURAL – DISSERTATION

zur
Erlangung der Doktorwürde
der
Gesamtfakultät für Mathematik, Ingenieur- und Naturwissenschaften
der
Ruprecht–Karls–Universität
Heidelberg

vorgelegt von Christoph Julian Klein
aus Menden (Sauerland)

Tag der mündlichen Prüfung ___________________

Algebraic, Block and Multiplicative Preconditioners based on Fast Tridiagonal Solves on GPUs

Advisor Prof. Dr. Robert Strzodka

Algebraic, Block and Multiplicative Preconditioners based on Fast Tridiagonal Solves on GPUs This thesis contributes to the field of sparse linear algebra, graph applications, and preconditioners for Krylov iterative solvers of sparse linear equation systems, by providing a (block) tridiagonal solver library, a generalized sparse matrix-vector implementation, a linear forest extraction, and a multiplicative preconditioner based on tridiagonal solves. The tridiagonal library, which supports (scaled) partial pivoting, outperforms cuSPARSE's tridiagonal solver by factor five while completely utilizing the available GPU memory bandwidth. The extraction of a weighted linear forest (union of disjoint paths) from a general graph is used to build algebraic (block) tridiagonal preconditioners and deploys the generalized sparse-matrix vector implementation of this thesis for preconditioner construction. During linear forest extraction, a new parallel bidirectional scan pattern, which can operate on double-linked list structures, identifies the path ID and the position of a vertex. The algebraic preconditioner construction is also used to build more advanced preconditioners, which contain multiple tridiagonal factors, based on generalized ILU factorizations. Additionally, other preconditioners based on tridiagonal factors are presented and evaluated in comparison to ILU and ILU incomplete sparse approximate inverse preconditioners (ILU-ISAI) for the solution of large sparse linear equation systems from the Sparse Matrix Collection. For all presented problems of this thesis, an efficient parallel algorithm and its CUDA implementation for single GPU systems is provided.

Algebraic, Block and Multiplicative Preconditioners based on Fast Tridiagonal Solves on GPUs Diese Promotion macht Beiträge im Bereich dünnbesetzter linearer Algebra, Graphanwendungen und Vorkonditionierer für iterative Krylov-Löser indem sie eine Bibliothek zum Lösen von (block-) tridiagonalen linearen Gleichungssystemen, eine Implementierung für ein generalisiertes dünnbesetztes Matrix-Vektor Produkt, eine Extraktion für lineare Bäume von Graphen und einen multiplikativen auf tridiagonalen Faktoren basierenden Vorkonditionierer vorstellt. Die tridiagonale Bibliothek unterstützt (skaliertes) partielles Pivoting und übertrifft die Performanz des Tridiagonallösers in cuSPARSE um Faktor fünf während die volle Bandbreite des GPU Speichers ausgenutzt wird. Die Extraktion der gewichteten linearen Bäume (Vereinigung von disjunkten Pfaden) von allgemeinen Graphen, wird verwendet um einen algebraischen (block-) tridiagonalen Vorkonditionierer zu konstruieren und verwendet dazu das generalisierte Matrix-Vektor Produkt dieser Thesis. Während der Baumextraktion, wird ein neuer paralleler bidirektionaler Scan verwendet um die ID und Position eines Knoten innerhalb eines Pfades zu bestimmen. Die Konstruktion des algebraischen Vorkonditioners wird ebenfalls verwendet um fortgeschrittene Vorkonditionierer zu konstruieren, die mehrere tridiagonale Faktoren enthalten und auf einer Generalisierung von ILU-Vorkonditionierern basieren. Zusätzlich evaluiert diese Thesis noch andere Vorkonditionierer, die auf tridiagonalen Systemen basieren und vergleicht diese mit ILU und ILU-ISAI Vorkonditionierern für die Lösung von großen dünnbesetzten linearen Gleichungssystemen aus der Sparse Matrix Collection. Für alle hier vorgestellten Problemklassen, stellt diese Thesis effiziente parallele Algorithmen und deren CUDA Implementierung für Computersysteme mit einer GPU vor.

Acknowledgements

I want to reveal my grateful acknowledgement regarding Prof. Dr. Robert Strzokda as my PhD advisor. I also thank my friends Marco, Julian, and Falk for their moral support. I thank my colleagues Moritz and Niklas for the good coffee breaks and technical discussions.

Contents

1. Introduction 13

2. tridigpu: A GPU library for block tridiagonal and banded linear equation systems 15
2.1. Introduction 15
2.2. Scalar Tridiagonal Solver Survey 18
2.2.1. Thomas Algorithm 18
2.2.2. Cyclic Reduction 19
2.2.3. Recursive Doubling 19
2.2.4. Parallel Cyclic Reduction 20
2.2.5. Partition Methods 20
2.2.6. Divide and Conquer 21
2.2.7. Redundant Reduction 21
2.3. Related Work 22
2.4. Contributions 22
2.4.1. Limitations 23
2.4.2. C API Overview 23
2.5. Recursive Partitioned Schur Complement Algorithm 24
2.5.1. Remark on Pivoting 26
2.5.2. Cyclic Systems 29
2.5.3. Block Tridiagonal Systems 29
2.5.4. Separate Factorization and Solve 30
2.6. Implementation 30
2.6.1. Active Warps 30
2.6.2. Data Layout 30
2.6.3. Shared Memory Layout 31
2.6.4. Storage Format for Row Interchanges during Pivoting 32
2.6.5. Complete Avoidance of SIMD Divergence 32
2.6.6. Avoidance of shared memory bank conflicts 32
2.6.7. Multiple Right-Hand Sides 32
2.6.8. Optimizations for small tridiagonal systems 33
2.6.9. Block Tridiagonal Systems 35
2.6.10. Factorization of a Block Tridiagonal Matrix 35
2.6.11. Banded Format to Block Tridiagonal Conversion 36
2.6.12. Calculation of $A^{-1}\mathcal{D}$ with Sparse Right-Hand Sides and Solutions 36
2.7. Results 37
2.7.1. Numerical Results 37
2.7.2. Performance Results 39
2.7.3. Calculation of $A^{-1}\mathcal{D}$ with Sparse Right-Hand Sides and Solutions 44
2.8. Summary and Outlook 48
2.9. Conclusion 48

3. Sparse Matrix-Vector Multiplication with Segmented Reduction 50
3.1. Introduction 50

3.2. Related Work . . . 51
3.3. Contributions . . . 52
3.4. Algorithms . . . 53
3.4.1. Preprocessing . . . 53
3.4.2. SRCSR . . . 56
3.5. Implementation . . . 56
3.5.1. Segmented Reduction Algorithms . . . 56
3.5.2. Warp Transposed Segmented Reduction . . . 56
3.5.3. Use of Atomic Operations . . . 57
3.5.4. Type Configuration . . . 57
3.5.5. Abstract Operators . . . 58
3.5.6. Auto-Tuning . . . 59
3.5.7. Partial Sparse Matrix-Vector Products . . . 59
3.6. Results . . . 60
3.6.1. Overall Performance . . . 60
3.6.2. General Applications of SRCSR . . . 62
3.7. Conclusion . . . 62
3.8. Other Contributors . . . 63

4. Highly Parallel Linear Forest Extraction from a Weighted Graph on GPUs 64
4.1. Introduction . . . 64
4.2. Related Work . . . 65
4.3. Algorithms . . . 66
4.3.1. Factor Notation . . . 66
4.3.2. $[0, n]$-Factor Algorithms on Weighted Graphs . . . 67
4.3.3. From a $[0, 2]$-Factor to a Linear Forest . . . 68
4.4. Implementation . . . 71
4.4.1. Parallel $[0, n]$-factor computation for $n \leq 4$ . . . 71
4.4.2. From a $[0, 2]$-Factor to a Linear Forest . . . 72
4.4.3. Permute and Extract a Linear Forest . . . 73
4.5. Results . . . 73
4.5.1. Weight Coverage Results . . . 74
4.5.2. Performance Results . . . 75
4.6. Application . . . 80
4.7. Conclusion . . . 81

5. Operator Split Preconditioners 82
5.1. Operator Splittings . . . 82
5.1.1. Introduction . . . 82
5.1.2. Outer Solver . . . 83
5.1.3. Outline and Contributions . . . 84
5.1.4. Related Work . . . 84
5.1.5. Generalization of ILU Factorizations . . . 86
5.1.6. Diagonal Preconditioners . . . 86
5.1.7. Invariance under Diagonal Scaling . . . 87
5.1.8. Sparsity Patterns . . . 87
5.1.9. Direct Construction . . . 88
5.1.10. Adaptive Construction . . . 89
5.2. Tridiagonal Splittings . . . 90
5.2.1. Iterative Decomposition into Linear Forests . . . 92

5.2.2. Direct Tridiagonal Construction . . . 92
5.2.3. Adaptive Tridiagonal Construction . . . 93
5.2.4. Preconditioning with Fused Permutations . . . 93
5.3. Results . . . 94
5.3.1. Overview of Solvers . . . 94
5.3.2. General Settings . . . 94
5.3.3. Discussion . . . 95
5.3.4. Comparison with Classical ADI Methods . . . 101
5.3.5. Comparison with exactly solved ILU(0) . . . 102
5.4. Conclusions . . . 103
5.5. Other Contributors . . . 103

6. Conclusion **104**

A. Bibliography **106**

A. Appendix **115**
A.1. Tridiagonal Library . . . 115
A.1.1. Naming Conventions . . . 115
A.1.2. Data Format and Layouts . . . 115
A.1.3. Resource Consumption . . . 116
A.1.4. C API . . . 116

List of Figures

1. Different phases of the Thomas Algorithm. The blue squares represent non-zero entries in the matrix, whereas white squares represent zero entries. 18
2. Reduction phases of the Cyclic Reduction. 19
3. Example of a partition method. 20
4. Matrix patterns of Redundant Reduction with Kogge Stone pattern (see Figure 5). 21
5. Kogge-Stone reduction scheme of Redundant Reduction (see Figure 4). Image taken from [31]. 21
6. Matrix graph representation and matrix patterns during different phases of RPTS assuming no row permutations took place. The partition size M is equal to 7 and the system size is $\hat{N} = 21$. 24
7. Size of the coarse system $N' = 2N/M$ relative to the fine system size N in dependency of the partition size M. The line representing $N' = N/M$ is for comparison. 24
8. Example for downwards oriented elimination during reduction step of RPTS with partial pivoting. 29
9. Data transposition in shared memory of RPTS. A band (e.g. a_i) is loaded coalesced into shared memory and subsequently processed sequentially by a thread; different threads appear in different colors. 31
10. Pivoting decisions represented as a binary tree and encoded as the bit pattern `0b000...1101`. Either the current equation j or the next equation $j + 1$ is used to eliminate the coefficient in column j. 31
11. Banded format to block tridiagonal format conversion. Each color refers to one band. Here, $n = 2$, $N = 14$, $\hat{N} = 7$, $k_l = 2$, and $k_u = 2$. 36
12. Single-precision performance of scalar tridiagonal solvers for matrix 1 from Table 4 with size N. Left: RPTS global memory throughput of the finest stage. Right: performance comparison to the tridiagonal solver from cuSPARSE. Missing data points indicate that the required memory exceeds the available GPU memory. 44
13. Single precison scalar tridiagonal solver performance for different RPTS final stage solvers. Left: the corresponding speedup of FFSK relative to a final stage solver with a single CUDA thread. 44
14. Single precison scalar tridiagonal solver performance with $n_r = 32$ right-hand sides and N unknowns. 45
15. Single precison scalar tridiagonal solver performance of `tridigpuSgtsv` and `tridigpuSgtsv_csc2dense` with varying number of right-hand sides and N unknowns. 45
16. Single precison block tridiagonal solver performance of `tridigpu` with varying number of right-hand sides and N unknowns. `bgtsv` repeatedly calculates the diagonalization, whereas `bgtfs` uses a given factorization of the tridiagonal system for the solution. 46
17. Single precison block tridiagonal factorization performance (`tridigpuSgtf`) in comparison to the solving step (`tridigpuSgtfs`, $n_r = 1$). 46

18. Single precision banded solver performance. For `tridigpuSgbsv`, the maximum equation throughputs are shown. 47
19. `gtsv_csc2csc` runtime in single precision for many sparse right-hand sides $\mathcal{D}$ from the Sparse Matrix Collection (Section A.1.4). 47
20. Small example for matrix row assignment to CUDA blocks in CSR not-null space with $w = 23$. The colors represent different CUDA blocks. 54
21. SRCSR flow chart for each CUDA block (SR = segmented reduction). . . 55
22. Warp transposed segmented reduction in three steps on small example CSR matrix from Figure 20 (warp size = 4). 57
23. SRCSR type configuration for generalized sparse matrix-vector products in case of CUB's segmented reduction algorithm. The right column denotes typical data locations in GPU memory and the arrows indicate the direction of type conversion. Each box represents a user configurable type. 58
24. SpMV performance comparison for operation $y = \beta y + \alpha Ax$ on single precision matrices from Table 11. Top: times without the SRCSR preprocessing. Bottom: with SRCSR preprocessing. The cuSPARSE times are equal in both plots. 61
25. Relative time for preprocessing of SRCSR SpMV for operation $y = \beta y + \alpha Ax$ on single precision matrices from Table 11. The time is relative to the SpMV + preprocessing time. 61
26. SRCSR-adaptive with different type and operator instantiations for four other use cases in comparison to standard SpMV $y = \beta y + \alpha Ax$. The single precision matrices are listed in Table 11. 63
27. Edge proposition and confirmation for a $[0, 2]$-factor (Algorithm 8, with $n = 2$, $k = 0$, $k_m = 0$) executed on a small graph. 68
28. Three steps (bottom to top) of a parallel bidirectional scan for the graph of Figure 27 with $N = 10$ vertices and 4 paths. Vertices of the same path are connected with green horizontal lines. Each step represents one kernel launch. 68
29. Performance results for one kernel execution of edge proposition according to Algorithm 8, Lines 14-22 with $k > 0$, $m = 1$, $k_m = 0$ and different n in comparison to SpMV algorithms. Except cuSPARSE SpMV, all schemes use our generalized sparse matrix-vector implementation. 77
30. Double-precision BiCGStab convergence results with our algebraically constructed scalar and 2x2 block tridiagonal preconditioner in comparison to a Jacobi and tridiagonal preconditioner based on the original vertex ordering. 78
31. Memory throughput statistics (top) for both bidirectional scan kernels and total runtimes (bottom) including steps (1), (2), (3) from Section 4.3.3. The speedup of the parallel version is written above the bars. 79
32. Time breakdown for $[0, 2]$-factor computation (Algorithm 8 with $M = 5, m = 5, k_m = 0, n = 2$), and the extraction of the linear forest (Section 4.3.3) to algebraically construct a tridiagonal preconditioner. The total absolute running time in milliseconds is written above the bars. 80
33. Subsequent segmentations (top to bottom) of a 2D isotropic grid (5pt stencil) without reoccuring edges ($\omega = 0$) in the left and reoccuring edges ($0 < \omega < 1$) in the right column. 91
34. Preconditioner setup times for each test matrix from Table 17. For matrix ML_GEER, the GPU memory was oversubscribed during the setup of MOS-d, which resulted in page evictions and migrations of CUDA managed memory, and thus significant higher setup times than for the other matrices. 96
35. Relative time of the preconditioner for one GMRES iteration (single precision). 96

36. GMRES relative forward error on logarithmic y-scale against number of iterations (top) and against time per non-zero (bottom). Setup times are not included. Where the curves end with a dot, the corresponding relative residual norm has fallen below 10^{-6}. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. The first 4 preconditioners are identical for $m = 1$, this common curve helps to compare them among each other. 97
37. GMRES relative forward error on logarithmic y-scale against number of iterations (top) and against time per non-zero (bottom). Setup times are not included. Where the curves end with a dot, the corresponding relative residual norm has fallen below 10^{-6}. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. The first 4 preconditioners are identical for $m = 1$, this common curve helps to compare them among each other. 98
38. GMRES relative forward error on logarithmic y-scale against number of iterations (top) and against time per non-zero (bottom). Setup times are not included. Where the curves end with a dot, the corresponding relative residual norm has fallen below 10^{-6}. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. The first 4 preconditioners are identical for $m = 1$, this common curve helps to compare them among each other. 99
39. GMRES relative forward error on logarithmic y-scale against number of iterations for operator split preconditioners with $m = 2$ in comparison to geometric ADI preconditioner $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ on 2D problems. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. 101
40. Full double-precision GMRES with forward relative error on logarithmic y-scale against number of iterations for operator split preconditioners (with a weight coverage of at least 99%) in comparison to an exactly solved ILU(0). Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. 102

List of Tables

1. Symbols and their meanings in this chapter. The upper part refers to the interface of `tridigpu`, whereas the lower part refers to internal library parameters. 18
2. Data reads and writes from global GPU memory for one stage and different kernels of `tridigpu<t>gtsv` in case of a scalar tridiagonal system. See Table 1 for the meaning of the variables. 33
3. Data reads and writes from global GPU memory for one stage and different kernels of `tridigpu<t>bgtf` and `tridigpu<t>bgtfs`. s_t is `sizeof(T)` and s_i is `sizeof(int64_t)`. See Table 1 for the meaning of the variables. . . 35
4. Scalar tridiagonal matrix collection for numerical-stability analysis (taken from [113]). MATLAB functions are used. Function `tridiag(a,b,c)` returns a tridiagonal matrix with main diagonal `b` and sub and superdiagonals `a` and `c`. $U(-1, 1)$ is the uniform distribution between one and minus one. All matrices are of the same size N. The condition number was calculated with the `JacobiSVD` function of the Eigen3 library for matrices of size $N = 512$. 38
5. Forward relative error of double-precision results for numerical-stability analysis of scalar tridiagonal solvers. 39
6. Pentadiagonal matrix collection for numerical-stability analysis. MATLAB functions are used. Function `pentadiag(dll, dl, d, du, duu)` returns a pentadiagonal matrix with main diagonal `d`, lower diagonals `dll`, `dl`, and upper diagonals `du`, `duu`. $U(-1, 1)$ is the uniform distribution between one and minus one. All matrices are of the same size $N = 512$. The condition number was calculated with the `JacobiSVD` function of the Eigen3 library. 40
7. Left part: Forward relative error of double-precision results for numerical stability analysis of pentadiagonal solvers. Right part: coarse system condition for RPTS with partial pivoting and without pivoting. The matrix ID refers to the matrices in Table 6. 41
8. Left part: Forward relative error of double-precision results for numerical stability analysis of 2x2 block tridiagonal solvers for matrices of size $N = 1024$. Right part: coarse system condition for RPTS with partial pivoting and without pivoting. Each matrix was created by transforming the scalar tridiagonal test matrices in Table 4 to 2x2 block tridiagonal form. 42
9. Left: properties of matrices from the Sparse Matric Collection [27]. Right: forward relative error of double-precision results for numerical stability analysis of banded solvers. 43
10. Matrices from the Sparse Matrix Collection [27] which are used as sparse right-hand sides $\mathcal{D}$. `max_element`(`nnz`$_i$) is the maximum number of non-zeros per row. 45
11. Test matrices which are either from the Sparse Matrix Collection [27] or taken from [72] (ANISO{1,2,3}). 60

12. Read and written buffers in global GPU memory for the implementation of the edge proposition, which is expressed as generalized sparse matrix-vector product. 72
13. Edge proposition for vertex 4 (-) of Figure 27 expressed as reduction along matrix row $(A')_{4,j}$ from left to right. The accumulator consists of two $(n = 2)$ sorted pairs $((A')_{i,j}, j)$. 73
14. Pattern symmetric test matrices which are either from the Sparse Matrix Collection [27] or taken from [72] (ANISO{1,2,3}). $\overline{\Delta(G)}$ denotes the mean degree of the graph G. 74
15. $[0, 2]$-factor computation on the undirected graph of the given matrix and their relative weight coverage (Eq. 23) after five iterations $M = 5$, $c_\pi(5)$, and the maximal $[0, 2]$-factor $c_\pi(M_{\max})$, which is reached after $M_{\max}$ iterations. 75
16. $[0, n]$-factors of underlying undirected graphs and their relative weight coverages (Eq. 23) after five iterations $M = 5$, $c_\pi(5)$, $k_m = 0$, $m = 5$ of the parallel (PAR) Algorithm 8, in comparison to the results of the sequential (SEQ) Algorithm 6. The coverage of the sub- and superdiagonal in the original ordering is shown by c_{id} (Eq. 24). The weight coverage of the algebraically constructed 2x2 block tridiagonal preconditioner is shown in the right part of the table (see Section 4.6). 76
17. Matrices from the Sparse Matrix Collection [27] (SMC), generated with MFEM [6], hypre [45], and self-created (A); horizontal grouping corresponds to Figures 36, 37, 38; $c_{\mathtt{diag}}$ is the diagonal weight coverage of the matrix (Eq. 54), $\bar{n}_r$ the mean and $\hat{n}_r$ the maximum number of non-zeros per row. 95

1. Introduction

In the first decades of processor development, performance was mainly improved by accelerating the frequency of the core, which results in an increased computational speed. This was comfortable for the programmer because a code did not require any changes: once written, it could be executed on newer hardware to speed up the runtime. However, hardware limitations like heat conduction, cooling, and semiconductor scales required the development of multiprocessors to fulfill Moore's law [87]. The multiprocessor was a turning point for the single processor paradigm, and other more heterogeneous computer systems, with e.g. graphics processing units (GPUs), emerged. This was uncomfortable for the programmer because now a code required changes to be executed efficiently on newer hardware. While making a parallel version of a sequential algorithm on CPUs is difficult, GPU programming is even more complex, and the development of fully optimized GPU kernels is a tedious process. Usually, the sequential algorithmic structure must completely be replaced with a new parallel formulation in combination with a tricky technical implementation to achieve maximum GPU performance. Nevertheless, the popularity of GPUs in heterogeneous compute clusters has been an unbroken growing trend for the last decades, and many of the most powerful supercomputers worldwide use GPUs as accelerators[1], which makes the necessity for fully optimized GPU kernels an up-to-date issue. GPUs are throughput optimized architectures, which comes at the cost of relaxed memory consistent models, and usually have thousands of cores with several streaming multiprocessors. Contrary CPUs are latency optimized architectures which usually have a few cores on each processor. Between GPUs and CPUs, the maximum hardware performance in terms of floating point operations per second, memory bandwidth, or energy efficiency is often found in GPU architectures, which makes them attractive accelerators. However, speedups of GPU accelerated programs which significantly exceed the actual hardware performance differences between CPU and GPU often indicate poorly CPU reference versions.

This thesis contributes with highly parallel algorithms and their efficient GPU implementations to the field of sparse linear algebra, graph computations, and preconditioners for iterative Krylov solvers. Problem statements of these fields can be found in many applications as essential building blocks, which make up a large part of the application's runtime. Thus, optimizing these building blocks to the full hardware capacity is of high relevance for many applications.

The rest of the thesis is structured as follows: In Chapter 2, a (block) tridiagonal library (`tridigpu`) for GPUs with (scaled) partial pivoting is presented, which solves scalar tridiagonal systems by utilizing the full GPU memory bandwidth. A tridiagonal system is a linear set of equations in which only the lower diagonal, the diagonal, and the upper diagonal have non-zero coefficients. `tridigpu` also solves block tridiagonal systems with small coefficient blocks of size $n \times n$ with $1 \leq n \leq 4$, efficiently, and includes optimizations for small tridiagonal systems & multiple (sparse) right-hand sides. Strengths of the library are the support for (scaled) partial pivoting and the full bandwidth utilization, whereas the underlying hierarchical algorithm is limited by the conditioning of the coarser systems

[1] `https://www.top500.org/`

in the hierarchy.

In Chapter 3, a generalized sparse matrix–vector (SpMV) implementation for GPUs (SRCSR) is presented, which uses segmented reduction algorithms as a building block. The challenge for highly parallel SpMV implementations is the work balancing of the irregularly distributed non-zero matrix coefficients to the GPU threads, which is often resolved by using specialized implementations depending on the sparsity pattern. SRCSR does not contain any specializations, which depend on the sparsity pattern, and also provides a general type and operator abstraction such that it can be used for much more problem statements beyond normal SpMV. Different SRCSR schemes, which make different trade-offs between work-balancing, additional kernel launch overhead, and type and operator generality, are evaluated with respect to large sparse matrices from the Sparse Matrix Collection [27].

Based on the SpMV kernel of Chapter 3, another contribution is made to the field of graph applications in Chapter 4 by a weighted linear forest extraction, which is an acyclic subgraph with maximum degree two, and weighted $[0,n]$-factor computations, which is a subgraph with maximum degree n. Linear forests can be represented with appropriate vertex permutations by a tridiagonal adjacency matrix. In that way, an algebraic tridiagonal preconditioner is constructed. Another algebraic 2×2 block tridiagonal preconditioner is constructed by computing a $[0, 1]$-factor, coarsen the graph by contracting the vertex pairs, and extracting the linear forest from the coarser graph. The (block) tridiagonal systems are solved with the `tridigpu` library from Chapter 2 and evaluated with a biconjugate gradient stabilized (BiCGStab) iterative solver. The field of preconditioners is an active research area [94], with the aim to reduce the iterations and time to the solution for iterative methods. Some preconditioner approaches make use of the underlying problem, and problem-specific preconditioners might be used for optimal convergence. However, the new preconditioners presented in this thesis are constructed without knowing the underlying problem but algebraically from the system matrix, which makes them applicable to many sparse matrices.

A more complex multiplicative preconditioner is presented in Chapter 5, which uses the graph algorithms from Chapter 4 for preconditioner construction and the tridiagonal solver from Chapter 2 to apply the preconditioner. This approach represents an improvement of simple scalar or block tridiagonal preconditioners and shows how multiple tridiagonal systems are combined into one single preconditioner application. However, the ansatz is not restricted to tridiagonal systems only but is rather a generalization of ILU factorizations to multiple factors.

Also note that every for every problem statement of this thesis a parallel algorithm and an efficient GPU implementation has been developed.

Regardless of the order in which the chapters are arranged, each chapter is self-contained and the reader might choose a different order. Most parts of the chapters have already been published before, which is indicated by an additional note at the beginning of each chapter.

2. tridigpu: A GPU library for block tridiagonal and banded linear equation systems

References: Most parts of this chapter were already published in [72] and [75]. The new contributions are the survey of scalar tridiagonal solvers in Section 2.2, Figure 7, Section 2.5.1 with Figure 8, and Algorithm 1 & 2, which are modified versions of the algorithms from [72] but with support for block tridiagonal systems.

Abstract In this thesis we present a CUDA library with a C API for solving block cyclic tridiagonal and banded systems on one GPU . The library can process block tridiagonal systems with block sizes from 1x1 (scalar) to 4x4 and banded systems with up to 4 sub- and superdiagonals. For the compute intensive block size cases and cases with many right-hand sides, we write out an explicit factorization to memory, however, for the scalar case the fastest approach is to only output the coarse system and recompute the factorization. Prominent features of the library are (scaled) partial pivoting for improved numeric stability, highest performance kernels, which completely utilize GPU memory bandwidth, and support for multiple sparse or dense right-hand side and solution vectors. The additional memory consumption is only 5% of the original tridiagonal system, which enables the solution of systems up to GPU memory size. The performance of the state-of-the-art scalar tridiagonal solver of cuSPARSE is outperformed by factor 5 for large problem sizes of 2^{25} unknowns, on a GeForce RTX 2080 Ti.

2.1. Introduction

`tridigpu` is a library for the solution of scalar and block tridiagonal linear equation systems with multiple right-hand sides

$$AX = D, \text{ with} \tag{1}$$

$$AX := \begin{pmatrix} b_0 & c_0 & & & & a_0 \\ a_1 & b_1 & c_1 & & & \\ & a_2 & b_2 & c_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & a_{\hat{N}-2} & b_{\hat{N}-2} & c_{\hat{N}-2} \\ c_{\hat{N}-1} & & & & a_{\hat{N}-1} & b_{\hat{N}-1} \end{pmatrix} \begin{pmatrix} x_{0,0} & \cdots & x_{0,n_r-1} \\ x_{1,0} & \cdots & x_{1,n_r-1} \\ x_{2,0} & \cdots & x_{2,n_r-1} \\ \vdots & \ddots & \vdots \\ x_{\hat{N}-2,0} & \cdots & x_{\hat{N}-2,n_r-1} \\ x_{\hat{N}-1,0} & \cdots & x_{\hat{N}-1,n_r-1} \end{pmatrix},$$

$$D := \begin{pmatrix} d_{0,0} & \cdots & d_{0,n_r-1} \\ d_{1,0} & \cdots & d_{1,n_r-1} \\ d_{2,0} & \cdots & d_{2,n_r-1} \\ \vdots & \ddots & \vdots \\ d_{\hat{N}-2,0} & \cdots & d_{\hat{N}-2,n_r-1} \\ d_{\hat{N}-1,0} & \cdots & d_{\hat{N}-1,n_r-1} \end{pmatrix},$$

and $A \in \mathbb{F}^{N \times N}$, small coefficient blocks $a_i, b_i, c_i \in \mathbb{F}^{n \times n}$, $x_i, d_i \in \mathbb{F}^{n \times 1}$, and $i = 0, \ldots, \hat{N}-1$, where $\mathbb{F}$ is the field of real $\mathbb{R}$ or complex numbers $\mathbb{C}$. Matrix A has $\hat{N}$ block rows with block size $n \in \mathbb{N}$ and $N = \hat{N}n$ scalar rows. The number of right-hand sides of the equation system is denoted with $n_r \in \mathbb{N}$.

Solving scalar tridiagonal systems ($n = 1$) is required in many applications, e.g. in electrodynamics [63], fluid dynamics [68, 60], or computer graphics [67], semicoarsening for multigrid solvers [101], and preconditioning for multigrid solvers [53]. The solution of block tridiagonal systems ($n > 1$) is required in computational finance [48], computational fluid dynamics [102], or image restoration problems [20].

Equation 1 encompasses multiple problem classes which `tridigpu` can solve efficiently. These problem classes guide our discussion of the C API, the algorithms, the implementations and the results throughout this chapter.

Problem Class Cyclic cyclic tridiagonal systems with multiple right-hand sides ($a_0, c_{\hat{N}-1} \neq 0$),

Problem Class Scalar scalar tridiagonal systems with multiple right-hand sides ($n = 1$),

Problem Class Block block tridiagonal systems with multiple right-hand sides ($n > 1$),

Problem Class DIA banded systems with multiple right-hand sides ($n > 1$),

Problem Class ScalarCSC2Dense scalar tridiagonal systems with multiple sparse right-hand sides, $AX = \mathcal{D}$, with a CSC encoded sparse matrix $\mathcal{D}$,

Problem Class ScalarCSC2CSC scalar tridiagonal systems with many sparse right-hand sides and a pruned incomplete result, $A\mathcal{X} = \mathcal{D}$, with CSC encoded large sparse matrices $\mathcal{D}$ and $\mathcal{X}$.

In the following we list one small example for each problem class. Obviously, one example cannot express all possible parameter variations within the corresponding problem class, but only highlight a certain characteristic aspect.

Example of Problem Class Cyclic is a system with a_0, $c_{\hat{N}-1} \neq 0$, $n_r = 1$, $N = 6$, and $n = 1$

$$AX = \begin{pmatrix} 7 & 13 & & & & 1 \\ 2 & 8 & 14 & & & \\ & 3 & 9 & 15 & & \\ & & 4 & 10 & 16 & \\ & & & 5 & 11 & 17 \\ 18 & & & & 6 & 12 \end{pmatrix} \begin{pmatrix} x_{0,0} \\ x_{1,0} \\ x_{2,0} \\ x_{3,0} \\ x_{4,0} \\ x_{5,0} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{pmatrix} = D. \tag{2}$$

Example of Problem Class Scalar is a system with $n_r = 3$, $N = 6$, and $n = 1$

$$AX = \begin{pmatrix} 6 & 12 & & & & \\ 1 & 7 & 13 & & & \\ & 2 & 8 & 14 & & \\ & & 3 & 9 & 15 & \\ & & & 4 & 10 & 16 \\ & & & & 5 & 11 \end{pmatrix} \begin{pmatrix} x_{0,0} & x_{0,1} & x_{0,2} \\ x_{1,0} & x_{1,1} & x_{1,2} \\ x_{2,0} & x_{2,1} & x_{2,2} \\ x_{3,0} & x_{3,1} & x_{3,2} \\ x_{4,0} & x_{4,1} & x_{4,2} \\ x_{5,0} & x_{5,1} & x_{5,2} \end{pmatrix} = \begin{pmatrix} 1 & 7 & 13 \\ 2 & 8 & 14 \\ 3 & 9 & 15 \\ 4 & 10 & 16 \\ 5 & 11 & 17 \\ 6 & 12 & 18 \end{pmatrix} = D. \tag{3}$$

Example of Problem Class Block is a system with $n_r = 1$, $N = 8$, $\hat{N} = 4$, and $n = 2$

$$AX = \left(\begin{array}{cc|cc|cc|cc} 13 & 15 & 29 & 31 & & & & \\ 14 & 16 & 30 & 32 & & & & \\ \hline 1 & 3 & 17 & 19 & 33 & 35 & & \\ 2 & 4 & 18 & 20 & 34 & 36 & & \\ \hline & & 5 & 7 & 21 & 23 & 37 & 39 \\ & & 6 & 8 & 22 & 24 & 38 & 40 \\ \hline & & & & 9 & 11 & 25 & 27 \\ & & & & 10 & 12 & 26 & 28 \end{array}\right) \begin{pmatrix} x_{0,0} \\ x_{1,0} \\ x_{2,0} \\ x_{3,0} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} = D. \tag{4}$$

Example of Problem Class DIA is a pentadiagonal system with $n_r = 1$, $N = 8$, $\hat{N} = 4$, and $n = 2$

$$AX = \left(\begin{array}{cc|cc|cc|cc} 13 & 15 & 29 & & & & & \\ 14 & 16 & 30 & 32 & & & & \\ \hline 1 & 3 & 17 & 19 & 33 & & & \\ & 4 & 18 & 20 & 34 & 36 & & \\ \hline & & 5 & 7 & 21 & 23 & 37 & \\ & & & 8 & 22 & 24 & 38 & 40 \\ \hline & & & & 9 & 11 & 25 & 27 \\ & & & & & 12 & 26 & 28 \end{array}\right) \begin{pmatrix} x_{0,0} \\ x_{1,0} \\ x_{2,0} \\ x_{3,0} \end{pmatrix} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \end{pmatrix} = D. \tag{5}$$

Example of Problem Class ScalarCSC2Dense is a system with $n_r = 6$, $N = 6$, $\hat{N} = 6$, $n = 1$, sparse matrix $\mathcal{D} \in \mathbb{R}^{6\times 6}$ and a dense solution

$$AX = \begin{pmatrix} 6 & 12 & & & & \\ 1 & 7 & 13 & & & \\ & 2 & 8 & 14 & & \\ & & 3 & 9 & 15 & \\ & & & 4 & 10 & 16 \\ & & & & 5 & 11 \end{pmatrix} \begin{pmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & x_{0,4} & x_{0,5} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} \\ x_{3,0} & x_{3,1} & x_{3,2} & x_{3,3} & x_{3,4} & x_{3,5} \\ x_{4,0} & x_{4,1} & x_{4,2} & x_{4,3} & x_{4,4} & x_{4,5} \\ x_{5,0} & x_{5,1} & x_{5,2} & x_{5,3} & x_{5,4} & x_{5,5} \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & & 4 & \\ 5 & & & 6 & & 7 \\ & & & & & \\ & & 8 & 9 & & \\ 10 & & & & & \\ 11 & 12 & 13 & & & \end{pmatrix} = \mathcal{D}. \tag{6}$$

Note, that sparse matrix $\mathcal{D}$ contains the right-hand sides and we calculate $X = A^{-1}\mathcal{D}$.

Example of Problem Class ScalarCSC2CSC is a system with $n_r = 6$, $N = 6$, $\hat{N} = 6$, $n = 1$, and sparse matrices $\mathcal{D}, \mathcal{X} \in \mathbb{R}^{6\times 6}$, which is inverted exactly, but the output is written in a sparse form by omitting small values

$$A\mathcal{X} = \begin{pmatrix} 6 & 12 & & & & \\ 1 & 7 & 13 & & & \\ & 2 & 8 & 14 & & \\ & & 3 & 9 & 15 & \\ & & & 4 & 10 & 16 \\ & & & & 5 & 11 \end{pmatrix} \begin{pmatrix} x_0 & & x_6 & & & \\ & & x_7 & & & \\ x_1 & x_4 & x_8 & & x_{11} & \\ x_2 & & & x_9 & & x_{12} \\ & & & & & \\ x_3 & x_5 & & x_{10} & & \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & & 4 & \\ 5 & & & 6 & & 7 \\ & & & & & \\ & & 8 & 9 & & \\ 10 & & & & & \\ 11 & 12 & 13 & & & \end{pmatrix} = \mathcal{D}. \tag{7}$$

The calculated sparse matrix $\mathcal{X} := \texttt{prune}(X, S_p)$ is the dense result $X := A^{-1}\mathcal{D}$, pruned to the sparsity pattern S_p (all coefficient outside S_p are discarded), The sparsity pattern S_p is chosen such that the maximum absolute values of the dense result are included in the sparse result.

The rest of the chapter is organized as follows: Section 2.2 gives an overview about the development of parallel scalar tridiagonal solvers and algorithms. Section 2.3 discusses related work and Section 2.4 gives an overview of the contributions of this chapter. Algorithmic intrinsics are presented in Section 2.5 and the implementation details in Section 2.6. We conclude the chapter with the numerical and performance results in Section 2.7.

Symbol	Meaning
A	block tridiagonal matrix or banded matrix of size $N \times N$
N	number of unknowns in the linear equation system
n	size of one coefficient block, $n = 1, 2, 3, 4$
$\hat{N}$	number of block rows in the linear equation system ($N = \hat{N}n$)
a_i	lower band of coefficient blocks in the block tridiagonal matrix
b_i	diagonal of coefficient blocks in the block tridiagonal matrix
c_i	upper band of coefficient blocks in the block tridiagonal matrix
k_l	number of lower bands in a banded matrix, $k_l = 0, 1, 2, 3, 4$
k_u	number of upper bands in a banded matrix, $k_u = 0, 1, 2, 3, 4$
n_r	number of right-hand sides of the linear equation system
D	dense right-hand sides; matrix of size $N \times n_r$
$\mathcal{D}$	sparse right-hand sides encoded as a CSC matrix of size $N \times n_r$
X	dense solution vectors; matrix of size $N \times n_r$
$\mathcal{X}$	sparse solution vectors encoded as a CSC matrix of size $N \times n_r$
S_p	sparsity pattern matrix of size $N \times n_r$
M	number of block rows per partition of the tridiagonal system
L	number of partitions of size M, which are processed by one CUDA block
$\hat{n}_r$	number of right-hand sides, which are kept simultaneously in on-chip memory
n'_r	number of right-hand sides per batch of function `tridigpu<t>gtsv_csc2csc`

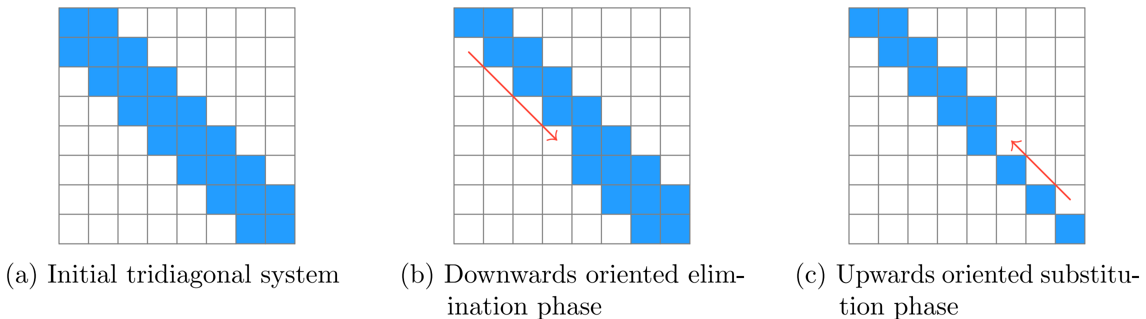
Table 1.: Symbols and their meanings in this chapter. The upper part refers to the interface of `tridigpu`, whereas the lower part refers to internal library parameters.

2.2. Scalar Tridiagonal Solver Survey

In this section an overview of the development of scalar tridiagonal solvers is given.

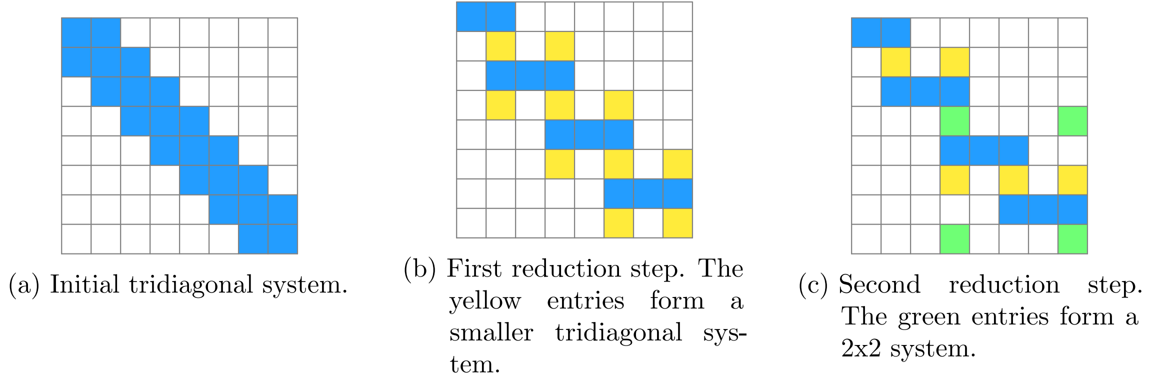
2.2.1. Thomas Algorithm

The Thomas Algorithm [109] represents the cassical sequential algorithm to solve tridiagonal systems. In fact, it is a systematic Gaussian elimination with a complexity of $\mathcal{O}(2N)$. In the first phase (Figure 1b) the entries a_i on the lower diagonal are eliminated and the values on the other bands (b_i and c_i) are modified. Afterward, the solution for x_{N-1} is known because the last row of the matrix contains only one non-zero value, which is the diagonal value. With the solution for x_{N-1}, the solution for x_{N-2} is calculated because the equation at row $N-2$ contains only two unknowns after the previous elimination



(a) Initial tridiagonal system (b) Downwards oriented elimination phase (c) Upwards oriented substitution phase

Figure 1.: Different phases of the Thomas Algorithm. The blue squares represent non-zero entries in the matrix, whereas white squares represent zero entries.



(a) Initial tridiagonal system.

(b) First reduction step. The yellow entries form a smaller tridiagonal system.

(c) Second reduction step. The green entries form a 2x2 system.

Figure 2.: Reduction phases of the Cyclic Reduction.

phase of the algorithm. In that manner all solutions for the x_i are computed (Figure 1c).

2.2.2. Cyclic Reduction

The Cyclic Reduction represents the first parallel algorithm to solve tridiagonal systems, and was first proposed by Hockney [63]. Similar to the Thomas Algorithm, it consists of a reduction and a substitution phase. After one reduction step, the tridiagonal system is reduced to a smaller system with half the size. In Figure 2b, a thread with index t operates on row $2t$, and uses row $2t+1$ and $2t-1$ to eliminate the dependency to x_{2t+1} and x_{2t-1}. During the elimination process additional entries for x_{2t-2} and x_{2t+2} are created. The yellow rows in Figure 2b form a smaller tridiagonal system, and the same reduction process is applied again (see Figure 2c) until the system has two unknowns only, which is solved directly. Afterward, two solution values of the small system are substituted in each equation of the larger system and the remaining value represents the divisor of the right-hand side to obtain the next solution value. This is shown in Figure 2c: once the solutions for the green system are calculated, they are substituted in the yellow system. A good overview of the access pattern is also explained by Kim et al. [69].

2.2.3. Recursive Doubling

Recursive Doubling (RD) was first proposed by [107], and Eğecioğlu et al. [43] reformulates the solution of a tridiagonal system as the calculation of a prefix sum with blocks $C_i, B_i \in \mathbb{C}^{3\times 3}$ defined as

$$C_0 := B_0, \tag{8}$$

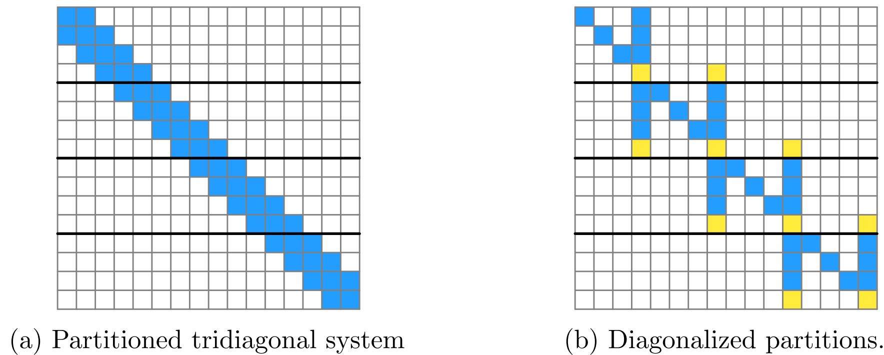
$$C_i := B_i C_{i-1}, \quad 1 \leq i \leq n-1, \tag{9}$$

and the recursively defined solution vectors

$$X_i := \begin{pmatrix} x_i \\ x_{i-1} \\ 1 \end{pmatrix} = C_{i-1} X_0, \tag{10}$$

where X_0 is calculated from C_{n-1}, and the B_i are constructed from the coefficients of equation i of the tridiagonal system. The Recursive Doubling allows the recursive solution of a tridiagonal system in three steps:

1. Compute all C_i with a parallel prefix sum (Equation 9).



(a) Partitioned tridiagonal system (b) Diagonalized partitions.

Figure 3.: Example of a partition method.

2. With C_{N-1}, obtain X_0.

3. Solve all X_i in parallel (Equation 10).

Since prefix sums are a well studied problem on parallel architectures, RD can be implemented using already existing algorithms. A disadvantage of this approach is that the prefix sum is calculated with 3×3 blocks,which must be saved in memory because these are required in the substitution phase again (step III).

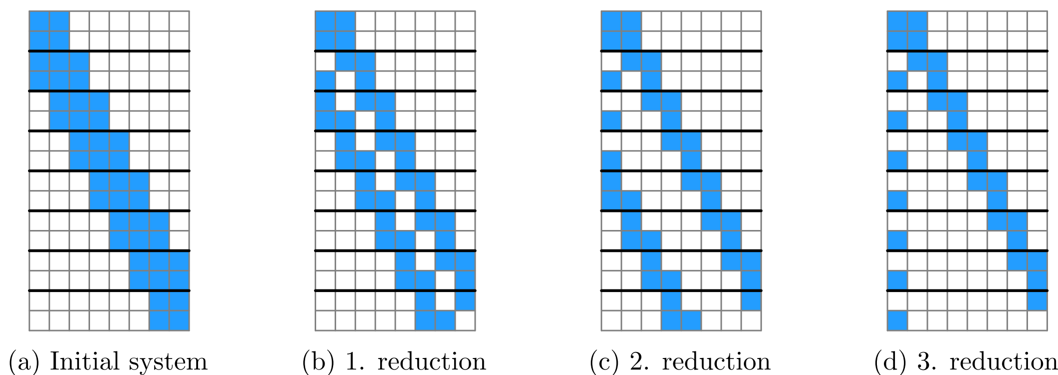
For a detailed explanation see [43], and for a short coneceptual one [112] or [122].

2.2.4. Parallel Cyclic Reduction

The Parallel Cyclic Reduction (PCR) is based on the Cyclic Reduction (CR) (see Section 2.2.2) and was also proposed by Hockney et al. [41]. Contrary to the Cyclic Reduction, the parallel version does not have a substitution phase. Typically a CR implementation assigns every second row to a thread, whereas a PCR implementation assigns each row to a thread, which does the elimination explained in Section 2.2.2. After the first reduction step, the original system of size N is reduced to two systems of size $N/2$. After thex next reduction step, there are four systems of size $N/4$. This pattern continues until there are N systems of size one, which are solved directly. A good overview of the access pattern is given by [69].

2.2.5. Partition Methods

Partition methods decompose the tridiagonal matrix into small partitions, which are diagonalized in parallel, with the cost of so called 'fill-ins'. E.g. Figure 3a shows a tridiagonal matrix of size $N = 16$, which is partitioned into 4 blocks of size 4. Next to that, Figure 3b shows the matrix after each block is diagonalized with Wang's approach [115]. The vertical spikes which occur for this diagonalization method can either be resolved sequentially, or the same technique can be applied to the smaller tridiagonal system, which is marked in yellow. Amodio et al. [4] formalized the concept of parallel factorization and Polizzi [100, 99] proposed a hybrid banded linear solver 'SPIKE' for parallel architectures. For the first time a numerically stable tridiagonal solver for the GPU was developed by Chang et al. [19, 18], which uses diagonal pivoting by Erway et al. [42]. Chang's implementation was improved regarding the numeric stability by Venetis et al. [113], which point out the problematic behaviour of diagonal pivoting in combination with singular submatrices and therefore proposed `g-Spike`.



(a) Initial system (b) 1. reduction (c) 2. reduction (d) 3. reduction

Figure 4.: Matrix patterns of Redundant Reduction with Kogge Stone pattern (see Figure 5).

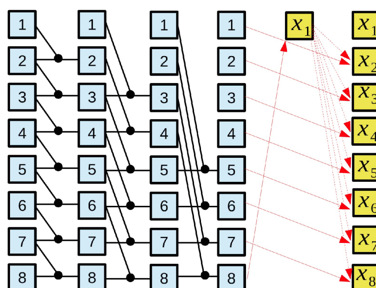


Figure 5.: Kogge-Stone reduction scheme of Redundant Reduction (see Figure 4). Image taken from [31].

2.2.6. Divide and Conquer

The Divide and Conquer approach by Wang et al. [116] is based on the idea that two adjacent N-shaped partitions (see Figure 3b) can be combined into one larger N-shaped partition. First, the last equation in the upper partition is used to eliminate the left fill-in of the lower partition. Second, The first equation of the lower partition is used to eliminate the right fill-in of the first partition. If one thread is assigned to one equation, the inter thread communication forms a critical performance bottleneck if the first and last equation is exchanged between the threads. Therefore, each thread saves the first and last equation of its partition redundantly.

The Divide and Conquer algorithm was implemented on a GPU by Lobeiras et al. [81] (BPLG) to solve small tridiagonal systems in shared memory. Moreover, Déguez et al. [32] developed an implementation for one GPU and benchmarked their application for problem sizes up to $2^{19} = 524288$ equations with an extension to larger system sizes for multiple GPUs in [96].

2.2.7. Redundant Reduction

The Redundant Reduction by Diéguez et al. [31] is used to solve small tridiagonal systems in GPU on-chip memory. As a first step, each equation E_i is split into two identical equations I_i^k and C_i^k, which doubles the size of the equation system (Figure 4). If two equations E_i and E_j are combined with the elimination operator, I_i^k and C_j^k have two variables in common, which can both be eliminated. Figure 4 shows that the rightmost variable of each equation stays fixed, whereas each reduction step shifts the most left x_i to

the left. In the last step, every equation contains one unknown, as the solution of the first variable is obtained from the last equation.

2.3. Related Work

On sequential architectures the Thomas Algorithm [109] is the classical choice to solve tridiagonal systems. Due to the increasing amount parallel computer architectures, parallel Algorithms like the Cyclic Reduction [63] (CR), the Parallel Cyclic Reduction [41] (PCR), the Partition Method [115], and the Divide and Conquer [116] approach were developed.

the Cyclic Reduction and the Parallel Cyclic Reduction were implemented by many authors [67, 26, 69, 53, 2] for a GPU. With the diagonal pivoting from Erway et al. [42], the first numerically stable tridiagonal solver for the GPU was implemented by Chang et al. [19, 18] and improved in aspects of numeric stability by Venetis et al. [113], who pointed out the problematic behaviour of diagonal pivoting and singular submatrices and therefore proposed `g-spike`. Two tridiagonal solvers were proposed by Diéguez et al. [31, 96, 32, 30, 29]: The Tree Partitioning Reduction (TPR) and a Wang and Mou implementation with their Butterfly Processing Library for GPUs (WM BPLG). The tridiagonal partitioning scheme deployed in this thesis, is also used by Giles et al. [50] and Lászlo et al. [77], to solve many independent small tridiagonal systems with a size of up to 1000 unknowns. The same partitioning scheme is also used by Klein and Strzodka [72] to solve large non-cyclic scalar tridiagonal systems with a single right-hand side, and Kim et al. [70], who proposed PaScaL_TDMA, which is a tridiagonal solver without pivoting for distributed memory machines, which communicate with MPI, to solve very large tridiagonal systems.

Batched block tridiagonal solvers [77] or batched scalar pentadiagonal solvers [49, 51] (PCR-Penta and cuPentBatch) for GPUs were proposed, but only Kamra and Rao [66] solved large block tridiagonal systems, who developed a parallel algorithm (DPBWZA) for block tridiagonal toeplitz-block-toeplitz linear systems. Ghosh and Mishra [49] proposed a Parallel Cyclic Reduction algorithm (PCR-Penta) to solve batched pentadiagonal systems. The above mentioned papers only treat special cases (batched, Toeplitz, pentadiagonal), whereas our `tridigpu` library solves a much more general problem (Eq. 1) including large cyclic block tridiagonal systems with multiple right-hand sides and support for (scaled) partial pivoting. Moreover, `tridigpu` solves general block tridiagonal systems without requiring symmetry and provides a comprehensive C API for multiple value types, as well as solving and factorizing operations.

2.4. Contributions

The contributions of this thesis regarding the tridiagonal solver library were published in [72, 75]. We developed a multi-functional GPU library (`tridigpu`) and made the following contributions:

- solution of scalar tridiagonal systems,
- support of different types of matrix coefficients and vector elements,
- solution of cyclic tridiagonal systems ($a_0 \neq 0$, $c_{\hat{N}-1} \neq 0$),
- multiple right-hand sides ($n_r > 1$),
- performance optimizations for small scalar systems ($N < 10^6$),

- block tridiagonal systems and their factorization ($n > 1$, $n_r \geq 1$),
- banded systems with up to four sub- and super-bands on each side of the diagonal,
- calculation of $A^{-1}\mathcal{D}$ for sparse matrices $\mathcal{D}$.

2.4.1. Limitations

Internally tridigpu uses the Recursive Partitioned Schur Complement Algorithm (RPTS) [72], in order to expose sufficient parallelism and solve large systems hierarchically. RPTS is based on the static partition methods, accuracy of which are limited by the conditioning of the coarse system, which is not controlled explicitly. Therefore, ill-conditioned coarse systems can negatively affect the quality of the calculated solution, although in practice this is a seldom occurrence. In case of scalar values, (scaled) partial pivoting works as expected by comparing the absolute values and implicitly permuting the rows. In case of small coefficient blocks (a_i,b_i,c_i), the determinants are compared and the block rows are implicitly permuted, but no row permutations within a block row occur. This imposes assumptions on the determinants of certain blocks and thus omits some opportunities for better numerical stability in favor of a unified code basis.

2.4.2. C API Overview

tridigpu follows the BLAS naming convention

tridigpu<t><format><operation>,

where <t> can be S, D, C, Z, which represent the types float, double, cuComplex, and cuDoubleComplex, respectively. As a placeholder for the corresponding types we use <T> and [c] denotes an optional character 'c' in a function name. <format> is either '[c]gt' for (cyclic) general tridiagonal systems, '[c]bgt' for (cyclic) block general tridiagonal systems, or 'gb' for general banded systems. <operation> is either 'sv' for a solving operation, 'f' for a factorizing operation, or 'fs' for a solving operation with a given factorization. tridigpu provides the following functions:

- **tridigpu<t>[c]gtsv:** This function solves Problem Class Cyclic and Problem Class Scalar, which are scalar (**c**yclic) **g**eneral **t**ridiagonal systems with multiple right-hand sides.
- **tridigpu<t>[c]bgtsv:** This function solves Problem Class Block, which are (**c**yclic) **b**lock **g**eneral **t**ridiagonal systems with multiple right-hand sides.
- **tridigpu<t>bgtf, tridigpu<t>bgtfs:** These functions solve Problem Class Block, where bgtf factorizes the matrix explicitly and bgtfs solves the system for a specific right-hand side and a given factorization.
- **tridigpu<t>gbsv:** This function solves Problem Class DIA, which are **g**eneral **b**anded systems with multiple right-hand sides.
- **tridigpu<t>gtsv_csc2dense:** This function solves Problem Class ScalarCSC2Dense, which are scalar tridiagonal systems with multiple sparse right-hand sides and dense solutions.
- **tridigpu<t>gtsv_csc2csc:** This function solves Problem Class ScalarCSC2CSC, which are scalar tridiagonal systems with many sparse right-hand sides and sparse solutions.

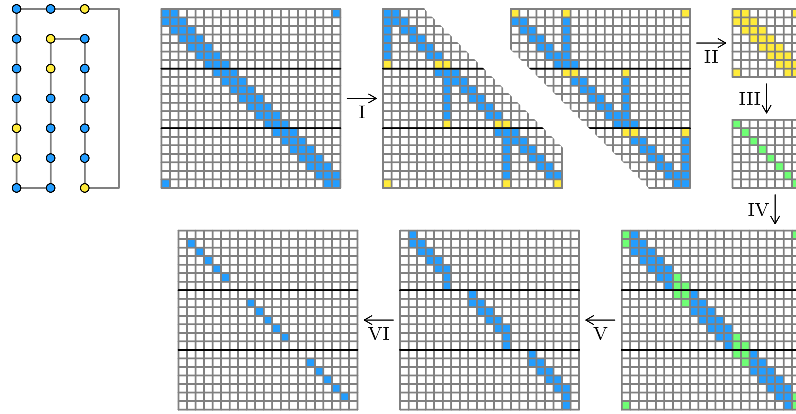


Figure 6.: Matrix graph representation and matrix patterns during different phases of RPTS assuming no row permutations took place. The partition size M is equal to 7 and the system size is $\hat{N} = 21$.

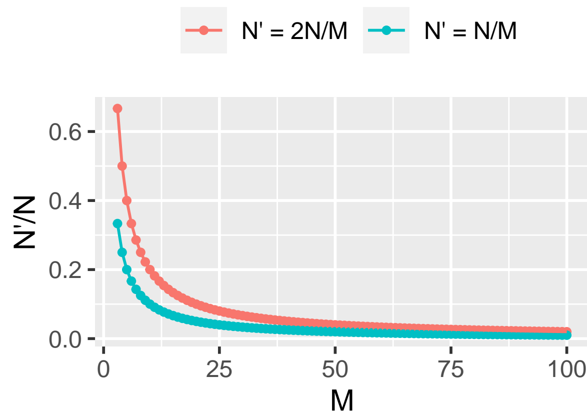


Figure 7.: Size of the coarse system $N' = 2N/M$ relative to the fine system size N in dependency of the partition size M. The line representing $N' = N/M$ is for comparison.

A detailed explanation of the C API is located in the Appendix A.1. The API design follows the conventions from cuSPARSE [90] and LAPACK [5], e.g., only the factorizing function `bgtf` overwrites the input tridiagonal system. Except `tridigpu<t>gtsv_csc2csc`, all function do not allocate any extra storage, enqueue the calculation into a CUDA stream, and are non-blocking, i.e., they may return control back to the host before the result is ready.

2.5. Recursive Partitioned Schur Complement Algorithm

During the work for this thesis, the **R**ecursive **P**artitioned **T**ridiagonal **S**chur Complement Algorithm with scaled partial pivoting was proposed [72] and extended [75] to solve Equation 1. Viewed as a graph a tridiagonal system A is a long chain of connected nodes as shown in Figure 6 on the left. We partition this chain into regular partitions of size M with 2 interface nodes and $M - 2$ inner nodes in each partition. Here, a node can be a scalar or a block matrix. Let I be the index set of all interface nodes and P be the index

set of all inner partition nodes. With respect to these index sets matrix A can be reordered by a permutation Q into the $(M-2)\times(M-2)$-block diagonal A_{PP}, the 2×2-block diagonal A_{II}, the wide A_{IP}, and the tall A_{PI}. The Schur-complement factorization then reads to:

$$QAQ^T = \begin{pmatrix} A_{PP} & A_{PI} \\ A_{IP} & A_{II} \end{pmatrix} = \begin{pmatrix} I & 0 \\ A_{IP}A_{PP}^{-1} & I \end{pmatrix} \begin{pmatrix} A_{PP} & A_{PI} \\ 0 & S \end{pmatrix}, \tag{11}$$

$$S := A_{II} - A_{IP}A_{PP}^{-1}A_{PI}, \tag{12}$$

with the Schur-complement S. In Equation 11, QAQ^T is a permuted form of A from Equation 1. The above factorization allows the recursive solution of

$$QAQ^T x = \begin{pmatrix} A_{PP} & A_{PI} \\ A_{IP} & A_{II} \end{pmatrix} \begin{pmatrix} x_P \\ x_I \end{pmatrix} = \begin{pmatrix} d_P \\ d_I \end{pmatrix} = d \tag{13}$$

in three steps

Reduce: Compute the coarse system S and the new right-hand side $d_I' := d_I - A_{IP}A_{PP}^{-1}d_P$.

Solve coarse system: Solve $Sx_I = d_I'$ by recursively applying the partitioning and factorization to S.

Substitute: Back-substitute x_I to obtain $d_P' := d_P - A_{PI}x_I$ and solve for the inner nodes $x_P := A_{PP}^{-1}d_P'$.

For a general A, the Schur-complement S has a different sparsity pattern than A, in fact S is usually dense. However, in this special case S is also tridiagonal, so we can recursively partition and factorize S in the same fashion as A until we arrive at a very small tridiagonal system which is solved directly. During the *reduction step* the inner of each partition is *diagonalized*, which produces fill-ins in the left-most and right-most columns, as seen after step I in Figure 6. For all partitions in parallel, the *downwards oriented elimination* of the lower band and the *upwards oriented elimination* of the upper band is realised by applying sequential elementary row operations with pivoting as shown in Figure 6 after step I. Our diagonalization of the reduction phase has an increased amount of parallelism because the upwards and downwards oriented elimination is calculated in parallel, which is indicated by the split matrix visualization in Figure 6.

After one reduction step, a smaller tridiagonal system is obtained with the size of $\hat{N}' = 2\hat{N}/M$, which is marked in yellow in Figure 6. In the following, the smaller system is referred to as the *coarse system* and the input system as the *fine system*. The reduction kernel of tridigpu is applied recursively, until the size of the system is sufficiently small, to be solved directly (step III in Figure 6).

The size of the coarse system is only a small fraction of the fine system, e.g. for $M = 37$ the size of coarse system is just 5% of the fine system, which can be seen in Figure 7. Thus from the computational point of view, increasing M further hardly yields any benefits. For comparison we plot the size of the coarse system if only one equation of each partition would contribute to the coarse system size, which shows only little improvements for large $M > 25$.

During the substitution phase, the solution of the coarse system is already known, and the corresponding values are substituted, which is shown in Figure 6 after step IV. The green values are known and can be substituted. In the substitution phase, the downwards oriented elimination is recalculated without fill-ins (step V) because the substitutions from the coarse system make each partition of the fine system independent of each other. The recalculation obviously incurs additional arithmetic operations, but we minimize the data

movement because neither the diagonalized system nor the permutation (pivot locations) must be written to memory after the reduction step. Since RPTS is still a bandwidth limited algorithm, trading additional computation for minimal data movement is beneficial. In particular, on GPUs with so many compute units, the additional calculations can be hidden behind the memory movement operations.

After step V in Figure 6, the solution of the $j = M - 2$ column is already available in each partition and is then substituted in the above equation to calculate the solution of the $j = M - 3$ column. This *upwards oriented substitution* continues in the same fashion until all solutions of the inner $M - 2$ block rows are computed (step VI). Steps IV, V and VI represent the *substitution step* of RPTS, which exposes less paralellism than the reduction step because the upwards oriented substitution is executed after the downwards oriented elimination.

Algorithm 1 sketches the calculation of the reduction step for one partition (see Figure 6); in the actual implementation many partitions are treated in parallel in this way. The indexing starts with zero and refers to the starting row of the current partition. a, b, c, d are vectors of length M, which represent the lower, middle, and upper band; and the right-hand side vector. s_c and s_p are temporary register arrays and denote the current and previous set of values, respectively ($s[0]$ is the spike, $s[4]$ is the right-hand side). r_c, r_p, m_c, and m_p are matrices of block size n. The function `reverse_view` reverses the view on the ranges a, b, c, or d (e.g. `reverse_view(`a`)`$[0] = a[M - 1]$). We use ';' as a delimiter between different instructions on the same line. ϵ is a threshold parameter, and $\tilde{\epsilon}$ is a diagonal matrix of size n with the smallest representable value in the current data format on the diagonal. The function `apply_threshold` maps blocks smaller than ϵ to zero blocks. This option allows the user to increase numeric stability in the case of noisy input values. Setting $\epsilon = 0$ switches off this behavior. In Line 43 and 44 the function `eliminate_band` is called in parallel for the upwards and downwards oriented elimination, which is possible because nothing needs to be saved in memory during the elimination.

Algorithm 2 shows the substitution step, which cannot execute the downwards oriented elimination and upwards oriented substitution in parallel. During the downwards oriented elimination, the row indices of the pivots must be saved on-chip so that the upwards oriented substitution finds the coefficients with respect to the correct pivot locations. As each interface has two nodes, the solution of $x[M - 2]$ and $x[1]$ can be obtained in two different ways, which is implemented in Lines 26-30 and Lines 38-42 of Algorithm 2, where the selection follows the partial pivoting criteria.

The pivoting of the Algorithms can be changed by choosing m_p and m_c accordingly:

no pivoting $m_p = m_c = 0$,

partial pivoting $m_p = m_c = \mathbb{1}$,

scaled partial pivoting shown in Algorithm 1 and 2.

2.5.1. Remark on Pivoting

Figure 8 shows an exemplaric downwards oriented elimination process during the reduction step of RPTS with implicit row permutations. Note that the coarse system is not built from the first and last row of a partition any more. The second row of the coarse system is not shown in the figure because it is calculated by the thread, which computes the upwards oriented elimination. During the upwards oriented substitution phase, some equations are unchanged with three unknowns and others only contain two unknows. No additional if-statement is required during the substitution phase in Algorithm 2 Line 39 because the

Algorithm 1: Reduction kernel of RPTS within a partition (step I & II, Figure 6).

```
1  Function apply_threshold(x, y)
2  |  if |det(x)| ≤ ϵ then  x = ϵ̃
3  |  if |det(y)| ≤ ϵ then  y = 0
4  end
5  Function get_max_abs(X_0, X_1, ..., X_{l-1})
6  |  P ∈ R^{n×n}
7  |  P = 0
8  |  for i = 0, ..., l − 1 do
9  |  |  for j = 0, ..., n − 1 do
10 |  |  |  for k = 0, ..., n − 1 do
11 |  |  |  |  P_jj = max(|(X_i)_jk|, P_jj)
12 |  |  |  end
13 |  |  end
14 |  end
15 |  return P
16 end
17 Function eliminate_band(a, b, c, d)
18 |  s_p[0] = a[1]; s_p[1] = b[1]; s_p[2] = c[1]
19 |  s_p[3] = 0; s_p[4] = d[1]
20 |  for j = 2, ..., M − 1 do
21 |  |  s_c[1] = a[j]; s_c[2] = b[j]
22 |  |  s_c[3] = c[j]; s_c[4] = d[j]
23 |  |  if (M = N̂) && is_cyclic && (j = M − 1) then
24 |  |  |  s_c[0] = c[M − 1]
25 |  |  else
26 |  |  |  s_c[0] = 0
27 |  |  end
28 |  |  apply_threshold(s_p[1], s_c[1])
29 |  |  m_p = get_max_abs(s_p[0], s_p[1], s_p[2])
30 |  |  m_c = get_max_abs(s_c[1], s_c[2], s_c[3])
31 |  |  if |det(m_p · s_c[1])| ≤ |det(m_c · s_p[1])| then
32 |  |  |  r_p = −(s_p[1])^{-1} s_c[1]; r_c = 1
33 |  |  else
34 |  |  |  r_p = 1; r_c = −(s_c[1])^{-1} s_p[1]
35 |  |  end
36 |  |  for k = 0, 2, 3, 4 do
37 |  |  |  s_p[k] = r_p s_p[k] + r_c s_c[k]
38 |  |  end
39 |  |  s_p[1] = s_p[2]; s_p[2] = s_p[3]; s_p[3] = 0
40 |  end
41 |  return s_p[0], s_p[1], s_p[2], s_p[4]
42 end
43 a_b, b_b, c_b, d_b = eliminate_band(a, b, c, d) // lower band
44 c_t, b_t, a_t, d_t = eliminate_band(reverse_view(c), reverse_view(b),
     reverse_view(a), reverse_view(d)) // upper band
45 write_coarse_system(a_b, b_b, c_b, d_b, a_t, b_t, c_t, d_t)
```

Algorithm 2: Substitution kernel of RPTS within a partition (step IV, V, and VI in Figure 6).

1 $d[M-2] = d[M-2] - c[M-2]\,x[M-1]$; $c[M-2] = 0$
2 $s_p[1] = b[1]$; $s_p[2] = c[1]$
3 $s_p[3] = 0$; $s_p[4] = d[1] - a[1]\,x[0]$; $i_p = 1$
4 **for** $j = 2, \ldots, M-2$ **do**
5 $\quad s_c[1] = a[j]$; $s_c[2] = b[j]$; $s_c[3] = c[j]$; $s_c[4] = d[j]$
6 $\quad$ `apply_threshold(`$s_p[1], s_c[1]$`)`
7 $\quad m_p =$`get_max_abs(`$s_p[1], s_p[2]$`)`
8 $\quad m_c =$`get_max_abs(`$s_c[1], s_c[2], s_c[3]$`)`
9 $\quad$ **if** $|$`det(`$m_p \cdot s_c[1]$`)`$| \leq |$`det(`$m_c \cdot s_p[1]$`)`$|$ **then**
10 $\qquad a[i_p] = s_p[1]$; $b[i_p] = s_p[2]$
11 $\qquad c[i_p] = 0$; $d[i_p] = s_p[4]$
12 $\qquad i[j-1] = i_p$
13 $\qquad r_p = -(s_p[1])^{-1} s_c[1]$; $r_c = 1$
14 $\qquad i_p = j$
15 $\quad$ **else**
16 $\qquad i[j-1] = j$
17 $\qquad r_p = 1$; $r_c = -(s_c[1])^{-1} s_p[1]$
18 $\quad$ **end**
19 $\quad$ **for** $k = 2, 3, 4$ **do**
20 $\qquad s_p[k] = r_p s_p[k] + r_c s_c[k]$
21 $\quad$ **end**
22 $\quad s_p[1] = s_p[2]$; $s_p[2] = s_p[3]$; $s_p[3] = 0$
23 **end**
24 $m_p =$`get_max_abs(`$s_p[1], s_p[2]$`)`
25 $m_c =$`get_max_abs(`$a[M-1], b[M-1], c[M-1]$`)`
26 **if** $|$`det(`$m_c \cdot s_p[1]$`)`$| \geq |$`det(`$m_p \cdot a[M-1]$`)`$|$ **then**
27 $\quad x[M-2] = (s_p[1])^{-1} s_p[4]$
28 **else**
29 $\quad x[M-2] = (a[M-1])^{-1}(d[M-1] - b[M-1]\,x[M-1] - c[M-1]\,x[M])$
30 **end**
31 **for** $j = M-3, \ldots, 2$ **do**
32 $\quad k = i[j]$
33 $\quad x[j] = (a[k])^{-1}(d[k] - b[k]\,x[j+1] - c[k]\,x[j+2])$
34 **end**
35 $k = i[1]$
36 $m_p =$`get_max_abs(`$a[k], b[k], c[k]$`)`
37 $m_c =$`get_max_abs(`$a[0], b[0], c[0]$`)`
38 **if** $|$`det(`$m_c \cdot a[k]$`)`$| \geq |$`det(`$m_p \cdot c[0]$`)`$|$ **then**
39 $\quad x[1] = (a[k])^{-1}(d[k] - b[k]\,x[2] - c[k]\,x[3])$
40 **else**
41 $\quad x[1] = (c[0])^{-1}(d[0] - b[0]\,x[0] - a[0]\,x[-1])$
42 **end**

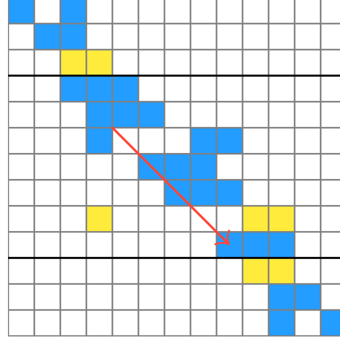


Figure 8.: Example for downwards oriented elimination during reduction step of RPTS with partial pivoting.

equations with two unknowns are shifted one column to the left in memory. Thus, the coefficient on the diagonal is saved on the lower diagonal and the coefficient of the upper diagonal is saved on the diagonal band.

2.5.2. Cyclic Systems

In case of cyclic tridiagonal systems ($a_0, c_{N-1} \neq 0$), the coarser system S of Equation 12 is also cyclic. During the RPTS reduction step (Figure 6, step II), the values a_0 and c_{N-1}, or due to pivoting their scaled values, are written to the upper right and lower left corner of the coarse equation system. In the higher stages, either the direct solver (step III) must be able to solve cyclic tridiagonal systems, or the system is reduced to a size of $N = 2$. In the latter case, the RPTS reduction step (Algorithm 1, Line 23) considers the values a_0 and c_{N-1} during the elimination of the last values because the fill-in and the a_0, c_{N-1} are located in the same column.

$$\text{downwards} \begin{pmatrix} b_0 & c_0 & & & & a_0 \\ a_1 & b_1 & c_1 & & & \\ l_2 & & b_2 & c_2 & & \\ l_3 & & & b_3 & c_3 & \\ l_4 & & & & b_4 & c_4 \\ c_5 & & & & a_5 & b_5 \end{pmatrix}, \begin{pmatrix} b_0 & c_0 & & & & a_0 \\ a_1 & b_1 & & & & r_1 \\ & a_2 & b_2 & & & r_2 \\ & & a_3 & b_3 & & r_3 \\ & & & a_4 & b_4 & c_4 \\ c_5 & & & & a_5 & b_5 \end{pmatrix} \text{upwards} \tag{14}$$

In a system, where no row permutations take place, and the last variables are about to be eliminated, which is a_5 in the downwards and c_0 in the upwards oriented elimination, as shown in Equation 14, the left fill-in l_4 refers to the same unknown as c_5.

2.5.3. Block Tridiagonal Systems

To solve block tridiagonal systems, the RPTS reduction and substitution Algorithms 1 and 2 support the non-commutativity of matrix-matrix and matrix-vector operations. Assume, there is the previous equation b_j, c_j and the current equation $a_{j+1}, b_{j+1}, c_{j+1}$, during the downwards oriented elimination, where b_j or a_{j+1} is about to be eliminated.

$$\begin{pmatrix} \ddots & \ddots & \ddots & & & \\ & 0 & b_j & c_j & & \\ & & a_{j+1} & b_{j+1} & c_{j+1} & \\ & & & \ddots & \ddots & \ddots \end{pmatrix} \tag{15}$$

For that purpose, the previous equation can be multiplied with $(-1)a_{j+1}b_j^{-1}$ and be added to the current equation, or the current equation can be multiplied with $(-1)b_j a_{j+1}^{-1}$ and be added to the previous equation. So implicitly we require that at least b_j or a_{j+1} is invertible.

Whereas the scalar pivoting decision is made based on the absolute value of a scalar, the pivoting for block types is made based on the absolute value of the determinant of the block, which is shown in Algorithm 1, Line 5. To make the pivoting decisions invariant to the scaling of the linear system by a diagonal matrix, the scaled partial pivoting decision is made with respect to the maximum absolute value within each row of the equation system (diagonal matrices m_p and m_c).

2.5.4. Separate Factorization and Solve

An explicit factorization of the tridiagonal matrix is beneficial, if the diagonalization cannot be hidden behind data movement any more, which is the case for block tridiagonal systems and multiple right-hand sides. After the downwards oriented elimination (step I, Figure 6), the calculated LU-factorization of the inner partitions $(L_{PP}^{(d)} + I)U_{PP}^{(d)} := A_{PP}$ is saved in memory, where $L_{PP}^{(d)}$ is strictly lower triangular and $U_{PP}^{(d)}$ upper triangular. As a consequence of saving the modified coefficients, the upwards and downwards oriented elimination are not executed in parallel, but in two subsequent steps: upwards, and then downwards. The latter only affects the order of computations and the coarse system S of the factorization is the same as that of Equation 12. During the upwards oriented elimination, which computes the UL-factorization $(U_{PP}^{(u)} + I)L_{PP}^{(u)} := A_{PP}$, with strictly upper triangular matrix $U_{PP}^{(u)}$ and lower triangular matrix $L_{PP}^{(u)}$, the coefficients of $L_{PP}^{(u)}$ are discarded, but $U_{PP}^{(u)}$ is kept, which is required to calculate the coarse right-hand side d'_I for a right-hand side d. Similar to the solve of scalar tridiagonal systems, the solution of the inner nodes x_P with a given factorization is obtained by $x_P := ((L_{PP}^{(d)} + I)U_{PP}^{(d)})^{-1}d'_P$.

2.6. Implementation

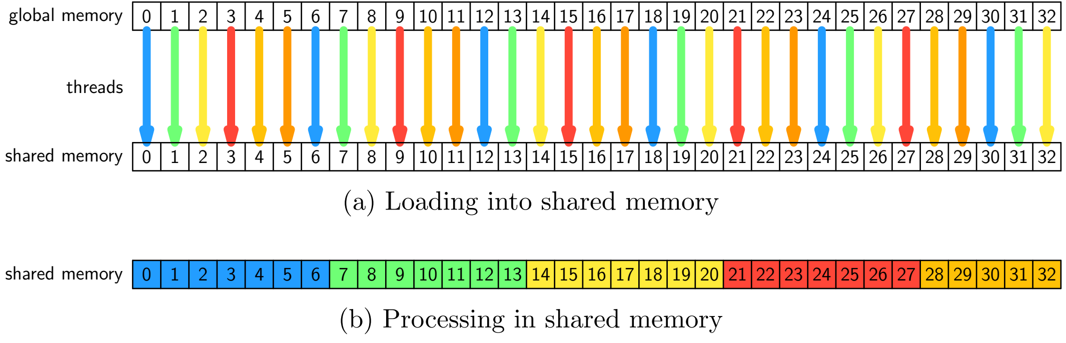
The core implementation of the `tridigpu<t><format>sv` functions, consists of two CUDA kernels: the *reduction kernel* and the *substitution kernel*, which compute the reduction and substitution step of RPTS (see Section 2.5) for arbitrary types of the matrix coefficients and vector elements. In fact, the templatized source code for solving the inner partitions is the same for all problem classes, whereas the source code, which loads the data to on-chip memory and writes the result to global memory, differs between the problem classes.

2.6.1. Active Warps

The number of processed partitions per CUDA block is usually $L = 32$ because we want a large M (shallow recursion) and the product LM is limited by the amount of available shared memory per block. For $L = 32$, there are two CUDA warps in the reduction kernel and one CUDA warp in the substitution kernel solving the inner partitions actively. The other threads in the CUDA block only participate during data load and store operations.

2.6.2. Data Layout

The block tridiagonal matrix is saved in a banded format, which are three buffers (a_i, b_i, c_i) of length $\hat{N}$. a_0 and $c_{\hat{N}-1}$ can be unequal to zero and represent the coefficients, which



(a) Loading into shared memory

(b) Processing in shared memory

Figure 9.: Data transposition in shared memory of RPTS. A band (e.g. a_i) is loaded coalesced into shared memory and subsequently processed sequentially by a thread; different threads appear in different colors.

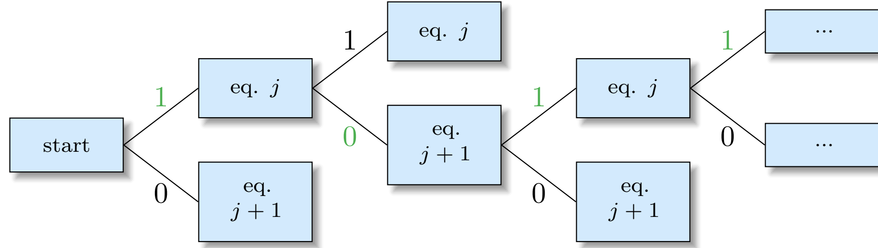


Figure 10.: Pivoting decisions represented as a binary tree and encoded as the bit pattern `0b000...1101`. Either the current equation j or the next equation $j+1$ is used to eliminate the coefficient in column j.

couple the first and last equation. The additional memory consumption of RPTS, which is required by the higher stages in the hierarchy, is very little, and for $n_r = 1$, $\hat{N} = 2^{25}$ or $\hat{N} = 2^{20}$ and $M = 41$ only $\approx 5\%$ of the input data.

2.6.3. Shared Memory Layout

To ensure full memory bandwidth the threads must load the bands and right-hand side of their partitions in a coalesced way into shared memory. Figure 9a shows an example for one thread block of 6 threads, which load their data into shared memory. Subsequently, Figure 9b demonstrates that during the elimination each thread processes the values of its partition (with size $M = 7$) sequentially. Thread 0 reads element 0 while thread 1 reads element 7. All threads participate in reading and writing data, but not necessarily all threads of a block participate in the elimination process due to the fact that the number of partitions per block L is another tuning parameter. Generally, $L = 32$ is already sufficient because then one full CUDA warp calculates the elimination. The example in Figure 9 shows that only 5 threads process a partition, although 6 threads load the data into shared memory. In general N is not divisible by M and the last thread in the last block, which participates in the elimination process, processes less than M equations.

In case of one right-hand side, the overall shared memory consumption per block is $(3Mn^2 + Mn)L$ elements, i.e., the three bands a, b, c plus right-hand side d. During the substitution phase it is necessary to save the solution x in shared memory, but only $2Ln$ additional shared memory elements must be occupied because x may reuse the buffer for the right-hand side.

2.6.4. Storage Format for Row Interchanges during Pivoting

The array, which saves pivot locations, denoted by $i[j]$ in Algorithm 2 requires M entries. Thus if the array is saved trivially as indices, additional memory of $M \cdot L$ integers would be required for each CUDA thread block. This could be saved in either shared memory or registers. If $i[j]$ were saved in shared memory, the overall shared memory consumption per CUDA thread block would increase unneccassarily limiting the maximum possible value for M. Saving $i[j]$ in registers, on the other hand, would decrease the occupancy and thus the latency hiding ability of the GPU. Therefore, we develop a minimal storage format for the pivot locations.

When a thread calculates the downwards oriented elimination, there are only two options, the current or the previous row is scaled and added to create a zero in the column j. Hence it is sufficient to use only one bit to distinguish between these two options, and M bits are required to save the pivoting locations for one partition. This is visualized as a binary tree in Figure 10. During the upwards oriented substitution, the actual index $i[j]$ is efficiently reconstructed from the bit pattern with bitwise operations. In the current implementation, a `int64_t` with 64 bits saves the bit pattern. This limits M to 64, however, larger values of M are not required, because the size of the coarse system is already negligible in comparison to the fine system (see Figure 7).

2.6.5. Complete Avoidance of SIMD Divergence

At first look, Algorithm 1 and 2 appear to have an overly complicated access to the bands and values of a, b, c, d through $s_p[0]$, $s_p[1]$, $s_p[2]$, $s_p[3]$, $s_c[0]$, $s_c[1]$, $s_c[2]$, $s_c[3]$. This is the result of a carefully crafted algorithmic formulation of the elimination in which all data-dependent conditionals assume the form of a value selection

```
result = condition ?  value1 :  value0
```

and only a little predication is required in the computation of `value1` and `value0`. Note, the writing of the coefficients in Lines 10 and 11 in Algorithm 2 can be placed in front of the if-statement at the cost of writing them redundantly in every iteration. Therefore, every if-statement is replaced with the aforementioned value selection in the implementation. The elimination of the coefficient in column j is then formulated as a linear combination (Alg. 1, Line 37) of two equations where the scaling factors of the equations depend on the pivoting decision. Consequently, the profiler reports zero SIMD divergence despite all threads making data-dependent pivoting decisions in parallel.

2.6.6. Avoidance of shared memory bank conflicts

The reduction kernel is completely free of shared memory bank conflicts because all pivoting decisions are processed in registers and the elimination of the lower and upper band is done by two different CUDA warps independently. If M is even, the shared memory arrays are padded by 1 ensuring zero bank conflicts in the parallel access. In the substitution kernel, bank conflicts cannot be avoided completely because the access to shared memory depends on the pivot locations.

2.6.7. Multiple Right-Hand Sides

The reduction kernel of `tridigpu` coarsens the tridiagonal system, until it is small enough to be solved directly. Subsequently, the substitution kernel recalculates the diagonalization of the partitions, but without any fill-in to obtain the fine solution vector. For the solution

kernel	read elements	written elements	read & written
reduction	$3N + n_r N$ $(A + D)$	$3\frac{2N}{M} + n_r \frac{2N}{M}$ (coarse A + coarse D)	$3N + n_r N + \frac{2N}{M}(3 + n_r)$
substitution	$\frac{n_r}{\hat{n}_r} 3N + n_r N + n_r \frac{2N}{M}$ $(A + D +$ coarse $X)$	$n_r N$ (fine X)	$\frac{n_r}{\hat{n}_r} 3N + 2n_r N + n_r \frac{2N}{M}$
total	$3N(\frac{n_r}{\hat{n}_r} + 1) + 2n_r N + n_r \frac{2N}{M}$	$\frac{2N}{M}(3 + n_r) + n_r N$	$3N(\frac{n_r}{\hat{n}_r} + 1 + n_r) + \frac{2N}{M}(2n_r + 3)$ (fine system + coarse system)

Table 2.: Data reads and writes from global GPU memory for one stage and different kernels of `tridigpu<t>gtsv` in case of a scalar tridiagonal system. See Table 1 for the meaning of the variables.

of one scalar tridiagonal system the solver reads and writes in total $9N$ elements on the first stage. This can be seen in Table 2, where $\hat{n}_r$ is the number of right-hand sides, which are kept simultaneously in on-chip memory, such that the amount of executed diagonalizations is reduced. Now setting $n_r = 1$, $\hat{n}_r = 1$, and neglecting the terms of the coarse system with factor $2N/M$, $9N$ elements are read and written in total. With a coarse system size of $2N/M$, the amount of data which is read and written in higher stages is of order $\sim 9 \cdot 2N/M = 18N/M$, which is one order of magnitude smaller than the fine system ($9N$) because typical values for M for a scalar tridiagonal system are in the range $31 \leq M \leq 41$. For multiple right-hand sides, the tridiagonal system is only loaded once into on-chip memory in the reduction kernel because the tridiagonal system is not changed in memory during the diagonalization. The same optimization cannot be applied to the substitution kernel because the tridiagonal system is modified during the diagonalization, but the tridiagonal system is only loaded once for $\hat{n}_r$ right-hand sides. In comparison to a naive repeated solver execution, which needs to process $E = n_r 9N$ elements, solving n_r right-hand sides with one single solver execution only processes $E' = 3N(n_r/\hat{n}_r + 1 + n_r)$ elements, which in relation to E is only

$$
\begin{aligned}
E'/E &= \frac{3N(n_r/\hat{n}_r + 1 + n_r)}{n_r 9N} \\
&= \frac{n_r 3N(1/\hat{n}_r + 1/n_r + 1)}{n_r 9N} \\
(\hat{n}_r = 4) \qquad &= \frac{1/4 + 1/n_r + 1}{3} \\
(n_r \gg 1) \qquad &\approx \frac{5/4}{3} \approx 42\%.
\end{aligned}
$$

2.6.8. Optimizations for small tridiagonal systems

For a static partition size M, we observe that solving small tridiagonal systems is not utilizing all available GPU resources, which is expected because M determines the work per CUDA thread and is chosen as a trade-off between available shared memory and number of stages. For smaller M, less shared memory is used per CUDA block, therefore more CUDA blocks can be resident per streaming multiprocessor, but the size of the coarser tridiagonal system N' increases because of $N' = 2N/M$. For larger M, the shared memory consumption is high, and less CUDA blocks are resident per streaming multiprocessor, which is inefficient in the limit of only one active CUDA block because for each CUDA block only one warp is calculating the tridiagonal factorization in the substitution kernel. To achieve maximum GPU performance it is essential to make use of latency hiding techniques, which is achieved by changing the context to another warp, if the current

warp is idling. The latter is impossible, if only one CUDA block is resident per streaming multiprocessor, thus, M must not be chosen too large. Therefore, our aim is to decrease the number of the stages and to increase the GPU utilization. To address the former point, we developed a fused final stage kernel (FFSK), which is able to solve tridiagonal systems of size $\texttt{warpSize} \cdot M$ on chip, and to address the latter point we use a dynamic choice of M, which depends on the GPU occupancy.

Fused Final Stage Kernel

The FFSK is a kernel fusion of the reduction kernel, single thread final stage kernel, and substitution kernel of RPTS. The single thread final stage kernel of RPTS [72] only uses one single CUDA thread to solve tridiagonal systems of maximum size $\tilde{N} = 32$. $\tilde{N}$ is the maximum tridiagonal system size of the final stage tridiagonal solver. The FFSK loads the tridiagonal system once into shared memory, calculates the RPTS reduction step, saves the coarse tridiagonal system in shared memory, solves the coarse system, calculates the RPTS substitution step to obtain the fine solution, and stores the fine solution back to global GPU memory. It is possible to use other well known parallel algorithms, like the (Parallel) Cyclic Reduction, as a final stage solver, but in our experiments we observed that these algorithms are less numerically stable. The FFSK is implemented with partial pivoting for improved numeric stability.

Algorithm 3: Dynamic choice of the partition size M_i for each stage i of RPTS.

input : N, sorted $\{\tilde{M}_j\}$, $\hat{n}_r$
output: $\{N_i\}$, $\{M_i\}$

1 $j_{\max} = \texttt{max}_j(\{\tilde{M}_j\})$
2 $\tilde{N} = \texttt{warpSize} \cdot \tilde{M}_{j_{\max}}$
3 $N_0 = N$
4 $i = 0$
5 $M_0 = \tilde{M}_{j_{\max}}$
6 **while** $N_i > \tilde{N}$ **do**
7     $j = j_{\max} - 1$
8   **while** *grid_dim*$(N_i, M_i) <$ *max_concurrent_blocks*$(N_i, M_i, \hat{n}_r) \ \wedge \ M_i \neq \tilde{M}_0$ **do**
9         $M_i = \tilde{M}_j$
10         $j = j - 1$
11   **end**
12     $i = i + 1$
13     $N_i = 2 \cdot \texttt{ceil}(N_{i-1}/M_{i-1}) - (N_{i-1} \ \% \ M_{i-1} \ == \ 1 \ ? \ 1 : 0)$
14     $M_i = \tilde{M}_{j_{\max}}$`// set M for next loop iteration`
15 **end**
16 $M_i = \texttt{get_smallest_final_stage_M}(N_i, \{\tilde{M}_j\})$

Dynamic Choice of M

Algorithm 3 shows the dynamic choice of the partition size for each stage i of RPTS, where we denote an array of elements with curly braces $\{\ \cdot\ \}$. The $\{\tilde{M}_j\}$ are the sorted partition sizes, whereas M_i is the partition size and N_i the tridiagonal system size for stage i. In Line 1 the index $j_{\max}$ of the largest admissible partition size is obtained. In

kernel	bytes read	bytes written
factorization bgtf	$3\hat{N}n^2 s_t$ (fine A)	$(3\hat{N}n^2 + 2\hat{N}n^2 + 3\frac{2\hat{N}}{M}n^2)s_t + 2\frac{\hat{N}}{M}s_i$ (factorized A + r_u and r_d+ coarse A + pivots)
reduction bgtfs	$(2\hat{N}n^2 + n\hat{N}n_r)s_t + 2\hat{N}/Ms_i$ (factorized A + fine D + pivots)	$n_r\frac{2\hat{N}}{M}ns_t$ (coarse D)
substitution bgtfs	$(\hat{N}n^2 + 3\hat{N}n^2 + n_r\hat{N}n + n_r\frac{2\hat{N}}{M}n)s_t + \frac{\hat{N}}{M}s_i$ (r_d + factorized A + fine D + coarse X + pivots)	$n_r\hat{N}ns_t$ (fine X)

Table 3.: Data reads and writes from global GPU memory for one stage and different kernels of tridigpu<t>bgtf and tridigpu<t>bgtfs. s_t is sizeof(T) and s_i is sizeof(int64_t). See Table 1 for the meaning of the variables.

Line 6, the while-loop processes every stage configuration until the system is small enough to be solved directly. In Line 8, a maximum value for M_i is chosen, such that the CUDA grid is at least as large as the maximum possible number of concurrently running blocks. Subsequently, in Line 13, the size of the coarser tridiagonal system is calculated, and the algorithm ends by calculating the smallest possible partition size for the final stage solver in Line 16.

2.6.9. Block Tridiagonal Systems

The kernels of tridigpu are implemented for general numerical types, which support +,-,*,/,abs. Commutativity for operator '*' is not required as this is not fulfilled for block types either. Operator '/' for block types is equal to the multiplication with the inverted block, and abs calculates the absolute value of the determinant of the block, which is required to make the pivoting decisions. In the kernel implementation, the pivoting scheme is passed in as a template parameter, which avoids any uneccessary operations at runtime. Due to generic programming and templates, no seperate code paths are required to support block tridiagonal systems. Instead of instantiating the kernels with scalar value types, they are instantiated with a block matrix of fixed size when solving block tridiagonal systems.

2.6.10. Factorization of a Block Tridiagonal Matrix

The computational effort of the kernels, which always calculate the factorization (functions tridigpu<t><format>sv), is increasing with $\mathcal{O}(n^3)$. Therefore, a large block size prevents the kernels from running at full GPU memory bandwidth. In that case, it is efficient to precalculate the tridiagonal matrix factorization, such that no block inversions, determinant calculations and matrix-matrix operations are calculated redundantly. The overall data movement is shown in Table 3. The factorization kernel calcuates the coarse system ($2\hat{N}/M$ blocks), the pivot decisions ($\hat{N}/M$ elements of type int64_t), the inverted blocks r_u,r_d ($2\hat{N}$ blocks), and the modified tridiagonal system after the downwards oriented elimination ($3\hat{N}$ blocks). With r_u and r_d we denote the row multiplication factors of the upwards and downwards oriented elimination, respectively. In the lower band of the factorized matrix, we save the inverted blocks a_i^{-1} as these values must be inverted in the upwards oriented substitution step of RPTS (Algorithm 2 in [72], Line 31) to obtain the solution values. Therefore, the substitution and reduction kernel, which solve a system with a given factorization, are completely free of block inversions.

Given a precomputed factorization, the reduction kernel only reads $2\hat{N}$ inverted blocks (r_u, r_d), the pivot decisions, which are efficiently saved in bits, and the right-hand sides with $\hat{N}n_r$ blocks; and only the coarse right-hand sides are written to global memory. The

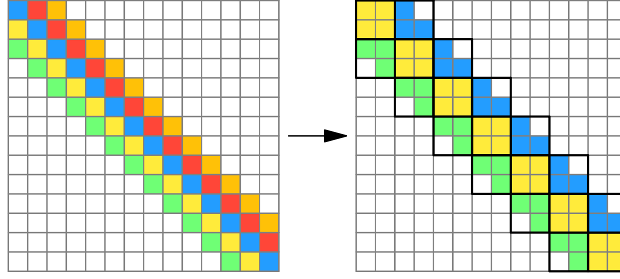


Figure 11.: Banded format to block tridiagonal format conversion. Each color refers to one band. Here, $n = 2$, $N = 14$, $\hat{N} = 7$, $k_l = 2$, and $k_u = 2$.

upwards and downwards oriented elimination is calculated in parallel and only consists of matrix-vector multiplications and vector additions. When we neglect the reading of the pivot decisions, the amount of read data is reduced by $\hat{N}$ for the reduction kernel of `bgtfs` by $\hat{N}$ elements in comparison to `bgtsv`. The substitution kernel of `bgtfs` reads one additional band of $\hat{N}$ elements in comparison to `bgtsv`, which are the inverted blocks r_d of the downwards oriented elimination, and the pivot decisions. The downwards oriented elimination must be calculated for the right-hand sides with the already available inverted blocks and pivot decisions. This cannot be pre-calculated in the reduction kernel because the solution from the coarse system is not available yet.

2.6.11. Banded Format to Block Tridiagonal Conversion

To solve banded systems with `tridigpu<t>gbsv`, the system is converted into block tridiagonal form while reading the bands from global to on-chip memory, as shown in Figure 11. Afterwards, the RPTS algorithm, can be used to solve the on-chip block tridiagonal system. To save a banded matrix with $k_l + k_u + 1$ bands as a block tridiagonal matrix, the blocks must be of of size $\texttt{max}(k_l, k_u)$, and for each matrix row $\texttt{max}(k_l, k_u) - 1$ zeros are additionally saved. Therefore, the memory consumption of a block tridiagonal matrix representation is increased by factor $1 + \frac{\texttt{max}(k_l,k_u)-1}{k_l+k_u+1}$ in comparison to the banded matrix representation. For full pentadiagonal systems ($k_l = k_u = 2$) this is an increase of 20%. Note that the block tridiagonal representation of the initial banded system is never saved in global GPU memory, but in on-chip memory. Only the intermediate results in the higher stages of the solver write and read block tridiagonal systems. However, this contributes little to additional data movement, because the coarse systems are small. Instead, the speed of the block tridiagonal solver is bounded by the limited thread parallelism in the computation.

2.6.12. Calculation of $A^{-1}\mathcal{D}$ with Sparse Right-Hand Sides and Solutions

Function `gtsv_csc2csc` is built of `gtsv_csc2dense` and a k-selection algorithm, which selects the k largest elements without sorting them with respect to their absolute value. The right-hand sides are processed in batches of n'_r vectors. For each n'_r right-hand sides, `gtsv_csc2dense` is called once, and the k-selection algorithm is called n'_r times, which results into $4n'_r$ histogram calculations in the worst case for single precision floating-point numbers. Our k-selection algorithm is similar to the `BucketSelect` of Alabi et al. [1], but has a fixed histogram size of 256 bins. We exploit that positive floating point numbers (IEEE 754-1989) remain their ordering when they are reinterpreted as unsigned integers based on their bit pattern. For single precision floating point numbers, a maximum of four

histograms are calculated to find the kth largest number. In the first histogram, each of the 256 bins represent the occurrence of a specific four byte prefix. Afterwards, one single CUDA block calculates the prefix sum and determines if an accumulated bin contains exactly k numbers. In that case, all numbers with the byte prefix represented by that bin are selected with CUB's `select_if` device primitive [84]. If there is no accumulated bin, which contains k numbers, the histogram and the prefix sum is calculated for the next four byte pattern until a bin contains the remaining number of required maximum elements, to obtain a byte prefix, which is used to select the k maximum values with CUB's `select_if` with respect to their absolute value.

For each right-hand side, which is processed by `gtsv_csc2csc`, k is set to the number of non-zero values in the corresponding sparse result matrix column.

2.7. Results

For the results presented in this chapter, we use a machine with a GeForce RTX 2080 Ti, CUDA 11.2.142, CentOS7, GCC 10.2.0, CUDA driver 450.57, and an Intel Xeon Silver 4110 @ 2.10 GHz. For performance results single-precision is chosen because the GeForce RTX 2080 Ti has very few double-precision arithmetic units. Throughout this section, we use partial pivoting if the pivoting scheme is not mentioned explicitly.

2.7.1. Numerical Results

There are four parameters, which control the numerical results of RPTS: First, the partition size M, second, the upper limit for the coarse system size $\tilde{N}$, which is solved directly (see Figure 6 step III), third, the threshold parameter ϵ, and fourth, the solver, which is used to solve the coarsest system directly. In the following results $\tilde{N} = 32$, $\epsilon = 0$ and a single CUDA thread with an adjusted version of Algorithm 2 is used to solve the coarsest system. The partition size is set to $Mn = 32$ block rows, such that there is only one reduction and one substitution step for the presented test cases.

For the evaluation, all calculations are done in double-precision and the solution vector x_t is generated with a normal distribution of floating-point numbers with a mean value of 3 and standard deviation of 1. The forward relative error is then calculated by $|x - x_t|_2/|x_t|_2$, where x is the calculated solution and x_t is the exact solution.

Scalar Tridiagonal Systems

RPTS as a scalar tridiagonal solver is compared against four other numerically stable tridiagonal solvers: First, cuSPARSE (`gtsv2`), which according to Venetis et al. [113] was a SPIKE implementation for GPUs by Chang et al. [19, 18] with diagonal pivoting [42]. Currently, the cuSPARSE documentation is not stating the specific algorithm, which is used by `gtsv2`, but the kernel names shown by the CUDA profiler, indicate that the aforementioned algorithm is still in use. Second, `g-spike` by Venetis et al. [113], which is an improvement regarding the numeric stability of the SPIKE implementation by Chang et al. [19]. Third, LAPACK's tridiagonal solver (`gtsv`). And fourth, the sparse LU decomposition implemented in Eigen3 [57].

Following the literature [113, 19, 30, 78], we use the matrices listed in Table 4 as scalar tridiagonal test cases for the numerical evaluation. All calculations are done in double-precision and the forward relative error is shown in Table 5.

ID	condition number	Description
1	1.58e+03	`tridiag(a, b, c)` with `a`, `b`, `c` sampled from $U(-1,1)$
2	1.00e+00	`b=1e+8*ones(N, 1)`; `a`, `c` sampled from $U(-1,1)$
3	3.52e+02	`gallery('lesp', N)`
4	2.93e+03	same as `#1`, but `a(N/2+1, N/2) = 1e-50*a(N/2+1,N/2)`
5	1.59e+03	same as `#1`, but each element of `a`, `c` has 50% chance to be zero
6	1.04e+00	`b=64*ones(N,1)`; `a`, `c` sampled from $U(-1,1)$
7	9.00e+00	`inv(gallery('kms', N, 0.5))` Toeplitz, inverse of Kac-Murdock-Szegö
8	1.02e+15	`gallery('randsvd', N, 1e15, 2, 1, 1)`
9	8.74e+14	`gallery('randsvd', N, 1e15, 3, 1, 1)`
10	1.11e+15	`gallery('randsvd', N, 1e15, 1, 1, 1)`
11	9.57e+14	`gallery('randsvd', N, 1e15, 4, 1, 1)`
12	3.07e+23	same as `#1`, but `a = a*1e-50`
13	1.40e+17	`gallery('dorr', N, 1e-4)`
14	8.17e+14	`tridiag(a, 1e-8*ones(N,1), c)` with `a,c` sampled from $U(-1,1)$
15	2.15e+20	`tridiag(a, zeros(N,1), c)` with `a,c` sampled from $U(-1,1)$
16	3.27e+02	`tridiag(ones(N-1,1), 1e-8*ones(N,1),ones(N-1,1))`
17	1.00e+00	`tridiag(ones(N-1,1), 1e8*ones(N,1),ones(N-1,1))`
18	3.00e+00	`tridiag(-ones(N-1,1), 4*ones(N,1),-ones(N-1,1))`
19	1.12e+00	`tridiag(-ones(N-1,1), 4*ones(N,1),ones(N-1,1))`
20	2.30e+00	`tridiag(-ones(N-1,1), 4*ones(N,1),c)`, `c` sampled from $U(-1,1)$

Table 4.: Scalar tridiagonal matrix collection for numerical-stability analysis (taken from [113]). MATLAB functions are used. Function `tridiag(a,b,c)` returns a tridiagonal matrix with main diagonal `b` and sub and superdiagonals `a` and `c`. $U(-1,1)$ is the uniform distribution between one and minus one. All matrices are of the same size N. The condition number was calculated with the `JacobiSVD` function of the Eigen3 library for matrices of size $N = 512$.

Block Tridiagonal and Banded Systems

In case of RPTS as a block tridiagonal and banded solver ($n = 2$), the results are compared, with well established and numerically stable CPU solvers: LAPACK (`gbsv`) and the sparse LU decomposition implemented in Eigen3 [57], as well as with the QR factorization implemented in cuSPARSE for the solution of batched pentadiagonal systems (`DgpsvInterleavedBatch`).

We create the 2x2 block tridiagonal test matrices by transforming the scalar tridiagonal test matrices from Table 4: If $A_s \in \mathbb{R}^{N/2 \times N/2}$ is a test matrix from Table 4, we obtain a 2x2 block tridiagonal test matrix $A_b \in \mathbb{R}^{N \times N}$ with the same condition as A_s by

$$(A'_b)_{ij} := \begin{cases} (A_s)_{\lfloor i/2 \rfloor, \lfloor j/2 \rfloor} & \text{if } i = j \\ (A_s)_{\lfloor i/2 \rfloor, \lfloor j/2 \rfloor - 1} & \text{if } i - 2 = j \\ (A_s)_{\lfloor i/2 \rfloor, \lfloor j/2 \rfloor + 1} & \text{if } i + 2 = j \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

$$A_b := O_l \, A'_b \, O_r, \tag{17}$$

where O_l and O_r are orthogonal 2x2 block diagonal matrices. With a uniformly distributed floating-point number $\alpha \in [0, \pi]$, a single 2x2 orthogonal block in O_l or O_r is generated by

$$\begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}. \tag{18}$$

We create the pentadiagonal test matrices analogously to the tridiagonal numerical test cases in Table 4, which are listed in Table 6.

matrix ID	Eigen3	RPTS	cuSPARSE	g-spike	LAPACK
1	5.72e-15	5.24e-15	5.05e-15	7.53e-15	5.78e-15
2	8.39e-17	8.32e-17	1.18e-16	1.30e-16	8.39e-17
3	1.28e-16	1.32e-16	1.44e-16	1.65e-16	1.29e-16
4	5.62e-15	5.25e-15	6.17e-15	1.55e-14	6.12e-15
5	1.19e-15	9.03e-16	1.94e-15	1.13e-15	8.85e-16
6	9.33e-17	9.57e-17	1.32e-16	1.50e-16	9.33e-17
7	2.33e-16	2.76e-16	2.53e-16	2.74e-16	2.34e-16
8	1.18e-04	4.53e-04	1.29e-05	5.52e-05	1.26e-04
9	4.01e-05	5.07e-05	2.77e-05	1.73e-05	5.73e-05
10	4.66e-05	1.25e-05	1.85e-05	4.88e-06	5.19e-05
11	5.35e-05	2.87e-04	1.46e-03	2.89e-03	3.57e-04
12	9.45e+03	1.35e+05	7.63e+05	2.51e+05	9.45e+03
13	1.08e+00	2.45e+00	1.33e+00	1.21e+00	4.37e-01
14	1.08e-03	1.76e-03	2.89e-03	9.05e-02	1.28e-03
15	5.21e+02	5.01e+02	9.24e+02	4.45e+02	5.21e+02
16	8.67e-16	1.37e-15	3.49e-15	3.89e-15	7.75e-16
17	1.14e-16	1.16e-16	1.60e-16	1.53e-16	1.14e-16
18	8.94e-17	1.04e-16	1.36e-16	1.42e-16	8.94e-17
19	1.10e-16	1.11e-16	1.51e-16	1.57e-16	1.10e-16
20	1.18e-16	1.11e-16	1.46e-16	1.51e-16	1.17e-16

Table 5.: Forward relative error of double-precision results for numerical-stability analysis of scalar tridiagonal solvers.

The double-precision results for 2x2 block tridiagonal matrices are shown in Table 8 and for pentadiagonal matrices in Table 7. The right part of each Table shows the conditioning of the coarse tridiagonal system, which has a size of $N' = \frac{2N}{Mn}$. The condition is `NA` if the coarse system could not be calculated due to numerical issues in the reduction step. The results of RPTS are comparable to that of Eigen3 and LAPACK, with exception of matrix 9 in Table 8; and matrix 12 and 17 in Table 7, which is caused by high coarse system condition numbers as a result of the static partitioning and blocking.

Additionally to the hard-to-solve synthetic problems, we show the forward relative error for real-world problems from the Sparse Matrix Collection in Table 9. Although the Sparse Matrix Collection contains many matrices, only few matrices have at most four sub- and super-bands on each side of the diagonal. In five cases, RPTS achieves a better error as LAPACK or Eigen3 but for matrices OLM1000 and SPMSRTLS, LAPACK and Eigen3 is clearly better (see Section 2.4.1).

2.7.2. Performance Results

Scalar Tridiagonal Systems

The reduction kernel of RPTS reads $4N$ elements, which are the three bands and the right-hand side, and writes $8N/M$ elements of the coarse system. The substitution kernel reads $4N + 2N/M$ elements, because only the solution of the coarse system is required, and writes the solution vector of length N. The throughput, which is shown in Figure 12, is measured with the `nvprof` for the GTX 1070 and with NVIDIA Nsight Compute for the RTX 2080 Ti. For comparison, the performance of a simple copy kernel, which just copies a certain amount of data is shown. Although the throughput of the copy kernel does not reach the theoretical memory bandwidth, the copy kernel performance usually displays the hardware performance limit for memory-bound algorithms. Whereas the copy kernel reads and writes N elements, this is different for the kernels of RPTS, which read more data than they write. Therefore higher throughputs than for the copy kernel are possible.

ID	condition number	description
1	3.45e+02	`pentadiag`($U(-1,1)$, $U(-1,1)$, $U(-1,1)$, $U(-1,1)$, $U(-1,1)$)
2	1.00e+00	`pentadiag`($U(-1,1)$, $U(-1,1)$, `1e8*ones(N)`, $U(-1,1)$, $U(-1,1)$)
3	3.45e+02	same as `#1`, but `dll[N/2+1] = 1e-50*dll[N/2+1]`
4	3.43e+02	same as `#3`, but `dl[N/2+1] = 1e-50*dl[N/2+1]`, `dll[N/2+2] = 1e-50*dll[N/2+2]`
5	3.43e+03	same as `#1`, but each element of `dll`, `duu` has 50% chance to be zero
6	7.78e+03	same as `#1`, but each element of `dll`, `dl`, `du`, and `duu` has 50% chance to be zero
7	1.06e+00	`pentadiag`($U(-1,1)$, $U(-1,1)$, `64*ones(N)`, $U(-1,1)$, $U(-1,1)$)
8	9.98e+14	`gallery('randsvd', N, 1e15, 2, 2, 2)`
9	9.43e+14	`gallery('randsvd', N, 1e15, 3, 2, 2)`
10	1.18e+15	`gallery('randsvd', N, 1e15, 1, 2, 2)`
11	9.65e+14	`gallery('randsvd', N, 1e15, 4, 2, 2)`
12	3.55e+07	same as `#1`, but `dll = dll*1e-50`
13	6.76e+34	same as `#12`, but `dl = dl*1e-50`
14	1.02e+01	`gallery('toeppen', N, 1, -10, 0, 10, 1)`
15	1.45e+03	`pentadiag`($U(-1,1)$, $U(-1,1)$, `1e-8*ones(N)`, $U(-1,1)$, $U(-1,1)$)
16	1.45e+03	`pentadiag`($U(-1,1)$, $U(-1,1)$, `zeros(N)`, $U(-1,1)$, $U(-1,1)$)
17	3.57e+04	`pentadiag(ones(N-2), ones(N-1), 1e-8*ones(N), ones(N-1), ones(N-2))`
18	1.00e+00	`pentadiag(ones(N-2), ones(N-1), 1e8*ones(N), ones(N-1), ones(N-2))`
19	2.56e+00	`pentadiag(-ones(N-2), -ones(N-1), 8*ones(N), -ones(N-1), -ones(N-2))`
20	2.56e+00	`pentadiag(-ones(N-2), ones(N-1), 8*ones(N), ones(N-1), -ones(N-2))`
21	1.51e+00	`pentadiag(-ones(N-2), ones(N-1), 8*ones(N), ones(N-1), ones(N-2))`
22	1.54e+00	`pentadiag(-ones(N-2), -ones(N-1), 8*ones(N), ones(N-1),` $U(-1,1)$)
23	1.85e+00	`pentadiag(-ones(N-2), -ones(N-1), 8*ones(N),` $U(-1,1)$, $U(-1,1)$)

Table 6.: Pentadiagonal matrix collection for numerical-stability analysis. MATLAB functions are used. Function `pentadiag(dll, dl, d, du, duu)` returns a pentadiagonal matrix with main diagonal `d`, lower diagonals `dll`, `dl`, and upper diagonals `du`, `duu`. $U(-1,1)$ is the uniform distribution between one and minus one. All matrices are of the same size $N = 512$. The condition number was calculated with the `JacobiSVD` function of the Eigen3 library.

In addition, the performance of the pure data movement without any calculation in the kernels of RPTS is shown for comparison. Processing the finest system heavily dominates the overall runtime. All coarse stages combined increase the overall runtime by only 8.5% for $N = 2^{25}$.

By comparing the kernels with and without calculation we see that for sufficiently large tridiagonal systems the computation is completely hidden behind data movement. Only for smaller problem sizes, the kernels of RPTS are slower than the data movement alone. The detailed throughput of RPTS was measured manually with CUDA events by using the mean kernel time of 700 kernel executions. All performance measurements use single-precision as the GTX/RTX graphics cards have only few double-precision units.

Figure 12 shows the equation throughput with a partition size of $M = 31$, $\epsilon = 0$, and a CUDA block dimension of 256. For large N, RPTS is approximately 5 times faster than the numerically stable tridiagonal solver of cuSPARSE on the RTX 2080 Ti. For further comparison, the non-pivoting version of cuSPARSE is also shown, which is a hybrid solver of CR and PCR. The absolute performance difference to cuSPARSE diminishes with decreasing N, which is partially caused by the decreasing memory throughput for small problem sizes. Although, this is a natural property of GPU memory, which can be seen by the copy kernel performance in Figure 12, the throughput measurements without any calculation denote the upper performance bound, and indicate that algorithms with higher computational parallelism can further improve the equation throughput, for small problem sizes.

ID	LAPACK	Eigen3	cuSPARSE	RPTS partial pivot	RPTS no pivot	κ_{partial} (coarse system)	κ_{nopivot} (coarse system)
1	2.92e-15	2.11e-15	1.00e-14	6.49e-15	2.98e-14	3.15e+02	2.46e+02
2	1.27e-16	1.27e-16	2.57e-16	1.31e-16	1.31e-16	1.00e+00	1.00e+00
3	2.81e-15	2.05e-15	6.97e-15	5.97e-15	2.86e-14	3.15e+02	2.46e+02
4	2.92e-15	2.07e-15	8.03e-15	5.92e-15	2.91e-14	3.15e+02	2.46e+02
5	5.13e-15	1.28e-14	1.60e-14	5.39e-14	1.85e-13	6.02e+02	6.02e+02
6	9.19e-15	9.19e-15	4.97e-14	8.85e-14	5.74e-13	7.70e+02	6.45e+03
7	1.43e-16	1.42e-16	3.85e-16	1.55e-16	1.55e-16	1.05e+00	1.05e+00
8	6.97e-04	6.97e-04	4.01e-03	2.99e-04	1.32e-01	2.18e+12	2.76e+15
9	8.43e-05	1.65e-04	3.19e-03	3.77e-04	1.69e-04	4.57e+14	1.85e+14
10	4.58e-04	4.06e-04	3.09e-03	2.60e-04	2.57e-04	4.24e+01	1.84e+01
11	7.31e-04	1.53e-03	1.18e-03	3.16e-04	2.12e-01	2.06e+14	2.08e+14
12	8.90e-12	8.90e-12	5.57e-11	5.75e-10	1.21e-10	1.84e+07	1.84e+07
13	2.44e+71	2.90e+71	6.86e+70	6.18e+72	8.24e+70	3.51e+28	5.12e+40
14	3.75e-16	3.65e-16	7.98e-16	3.62e-16	3.62e-16	2.94e+00	2.94e+00
15	9.23e-15	3.31e-15	9.59e-15	1.56e-14	2.39e-13	6.14e+02	2.89e+03
16	4.40e-15	4.54e-15	2.45e-14	7.39e-15	1.62e-13	6.14e+02	2.89e+03
17	5.97e-14	9.20e-14	3.35e-13	7.93e-04	4.96e+00	6.56e+08	4.49e+10
18	1.64e-16	1.63e-16	5.02e-16	1.40e-16	1.40e-16	1.00e+00	1.00e+00
19	2.00e-16	1.99e-16	7.33e-16	1.51e-16	1.51e-16	1.86e+00	1.86e+00
20	1.64e-16	1.65e-16	4.71e-16	1.63e-16	1.63e-16	1.86e+00	1.86e+00
21	1.63e-16	1.65e-16	4.47e-16	1.59e-16	1.59e-16	1.32e+00	1.32e+00
22	1.55e-16	1.55e-16	4.09e-16	1.51e-16	1.51e-16	1.31e+00	1.31e+00
23	1.49e-16	1.49e-16	3.53e-16	1.40e-16	1.40e-16	1.51e+00	1.51e+00

Table 7.: Left part: Forward relative error of double-precision results for numerical stability analysis of pentadiagonal solvers. Right part: coarse system condition for RPTS with partial pivoting and without pivoting. The matrix ID refers to the matrices in Table 6.

Optimizations for Small Tridiagonal Systems

Figure 13 displays the performance improvement of the fused final stage kernel (FFSK) and the dynamic choice of M in comparison to a final stage solver with a single CUDA thread and a static choice of M. As possible parameters for M we choose $\widetilde{M}_i = 17, 21, 31, 41$. As expected, the speedup converges against 1, for large problem sizes because then the reduction and substitution step of RPTS of the initial tridiagonal system dominate the runtime. Moreover, the cost of the cyclic final stage is shown, which reduces the tridiagonal system through additional stages down to a 2x2 matrix.

Multiple Right-Hand Sides

Figure 14 shows the performance of `tridigpuSgtsv` for a CUDA block dimension of 128, $n_r = 32$ right-hand sides, $M = 31$, and $\hat{n}_r = 4$ in dependence on the system size N and in comparison to the tridiagonal solvers of cuSPARSE (`gtsv2` and `gtsv2_nopivot`), which also support multiple right-hand sides. The equation throughput in [MRows/s] is calculated by $\frac{n_r N \cdot 10^{-6}}{\text{runtime}}$. The optimal choice of $\hat{n}_r$ and M depends on the CUDA architecture and is mainly affected by the amount of available shared memory.

Figure 15 shows that solving multiple right-hand sides within the same kernel is highly beneficial: `tridigpuSgtsv` for one right-hand side ($n_r = 1$) achieves ≈ 12500 MRows/s, but for $n_r = 32$ it runs at ≈ 31500 MRows/s for large N. The performance increase from $n_r = 1$, to $n_r = 2$, and to $n_r = 4$ is much larger than from $n_r = 4$ to $n_r > 4$ due to the caching of $\hat{n}_r = \min(n_r, 4)$ right-hand sides in on-chip memory. The dotted lines show the further increase in the performance for sparse right-hand sides with 10 non-zero

ID	LAPACK	Eigen3	RPTS partial pivot	RPTS no pivot	κ_{partial} (coarse system)	κ_{nopivot}
1	7.41e-15	5.80e-15	1.35e-14	1.26e-14	7.20e+02	4.68e+03
2	1.57e-16	1.57e-16	1.64e-16	1.64e-16	1.00e+00	1.00e+00
3	1.96e-16	1.90e-16	1.95e-16	1.95e-16	2.63e+02	2.63e+02
4	7.81e-15	6.80e-15	1.37e-14	1.35e-14	7.20e+02	8.75e+03
5	1.84e-15	2.54e-15	2.54e-15	9.33e-15	1.46e+02	2.33e+02
6	1.63e-16	1.54e-16	1.71e-16	1.71e-16	1.03e+00	1.03e+00
7	2.66e-16	2.74e-16	2.65e-16	2.65e-16	3.00e+00	3.00e+00
8	1.69e-04	1.99e-04	1.47e-04	3.04e-04	7.41e+12	2.51e+11
9	1.87e-05	5.08e-05	1.84e-02	6.10e-02	1.77e+15	5.36e+19
10	9.37e-05	1.22e-04	9.97e-05	6.24e-05	1.00e+00	6.14e+01
11	2.66e-03	3.14e-03	2.37e-03	1.77e-03	1.66e+14	1.32e+14
12	1.51e+06	9.51e+05	7.08e+05	5.11e+05	9.69e+22	2.48e+23
13	4.67e+00	1.88e+00	3.11e+00	2.83e+00	6.16e+17	5.74e+17
14	5.89e-03	6.91e-03	3.36e-03	7.11e+00	4.96e+14	3.23e+18
15	1.56e+02	3.34e+02	3.71e+02	NA	1.50e+20	NA
16	1.77e-15	2.79e-15	2.57e-15	4.53e-09	4.13e+01	4.13e+01
17	1.68e-16	1.58e-16	1.74e-16	1.74e-16	1.00e+00	1.00e+00
18	1.78e-16	1.77e-16	1.89e-16	1.89e-16	1.73e+00	1.73e+00
19	1.85e-16	1.88e-16	1.81e-16	1.81e-16	1.03e+00	1.03e+00
20	1.77e-16	1.77e-16	1.68e-16	1.68e-16	1.64e+00	1.64e+00

Table 8.: Left part: Forward relative error of double-precision results for numerical stability analysis of 2x2 block tridiagonal solvers for matrices of size $N = 1024$. Right part: coarse system condition for RPTS with partial pivoting and without pivoting. Each matrix was created by transforming the scalar tridiagonal test matrices in Table 4 to 2x2 block tridiagonal form.

entries, and dense solution vectors, is shown (`tridigpuSgtsv_csc2dense`). As expected, the reduction in data movement speeds up the runtime of the kernels. If a CUDA block detects that all right-hand side values are zero within its part of the tridiagonal system, the calculation is skipped and zeros are written to the solution vector immediately.

Block Tridiagonal Systems and their Factorizations

Analogously to the scalar tridiagonal solver performance analysis for multiple right-hand sides in Section 2.7.2, Figure 16 shows the performance of `tridigpu` block tridiagonal solvers. As expected, solving the system with the already factorized tridiagonal system achieves higher equation throughputs for increasing block sizes because the computationally expensive block inversions and determinent calculations have already been performed in the factorization. In this benchmark, solving the system with the previously calculated factorization is efficient for $\{n = 2, n_r \geq 4\}$, $\{n = 3, n_r \geq 4\}$, and $\{n = 4, n_r \geq 1\}$. The performance of the factorization (`gtf`) in comparison to the solving step (`gtfs`) with $n_r = 1$ is shown in Figure 17. As expected, the runtime difference increases for increasing block sizes n.

Banded Systems

For optimal performance results on the RTX 2080 Ti, we choose a CUDA block dimension of 128, $\{n = 2, M = 13, L = 32\}$, $\{n = 3, M = 7, L = 32\}$, and $\{n = 4, M = 7, L = 16\}$, where L denotes the number of partitions per CUDA block. For banded or block tridiagonal systems, M is rather small, due to the shared memory consumption of each block growing quadratically with the block size n. For $n = 4$, only half a warp is calculating the

matrix	problem	DOFs	nnz	bands	LAPACK	Eigen3	RPTS partial pivot	RPTS no pivot
NOS2	structural	957	4 137	9	1.90e+00	9.85e-01	2.10e+00	1.03e+01
OLM1000	fluid dynamics	1 000	3 996	7	8.90e-15	8.12e-15	1.01e-13	9.92e-14
OLM2000	fluid dynamics	2 000	7 996	7	1.73e-09	1.73e-09	6.92e-09	2.55e-09
OLM500	fluid dynamics	500	1 996	7	6.27e-13	1.20e-12	4.86e-14	3.29e-14
OLM5000	fluid dynamics	5 000	19 996	7	2.43e-12	2.82e-12	7.13e-13	7.73e-13
TUB1000	fluid dynamics	1 000	3 996	5	1.45e-13	1.35e-13	3.89e-14	1.13e-14
LINVERSE	statistical	11 999	95 977	9	1.32e-11	1.46e-11	2.27e-12	2.29e-12
SPMSRTLS	statistical	29 995	229 947	9	1.50e-14	2.67e-14	9.62e-11	9.62e-11
LF10000	model reduction	19 998	99 982	7	5.29e-12	3.33e-12	3.18e-13	8.52e-13

Table 9.: Left: properties of matrices from the Sparse Matric Collection [27]. Right: forward relative error of double-precision results for numerical stability analysis of banded solvers.

diagonalization in the substitution kernel, whereas still two half warps are active in the reduction kernel, due to the parallel upwards and downwards oriented elimination. We compare the performance against LAPACK's `gbsv` running on the CPU, and against the interleaved batched pentadiagonal solver (`SgpsvInterleavedBatch`) as well as the ILU(0) solver from cuSPARSE. The ILU(0) solver for fully populated banded systems, is identical to the LU factorization because no additional fill-in occurs. The performance shown for the ILU(0) does not include the time which is required for the calculation of the L und U factors. `SgpsvInterleavedBatch` is optimized for many small pentadiagonal systems, which fit into on-chip memory, the banded solve of `tridigpu`, can also solve many small systems when batched together into one, but can also solve very large pentadiagonal systems. Hence, this comparison is in favor of `SgpsvInterleavedBatch`, but this is the only other GPU library with explicit support for at least certain pentadiagonal systems. Figure 18 shows the achieved equation throughput for different solvers, number of bands in the matrix and tile sizes. The tile size only affects the performance of the cuSPARSE solvers, and not surprisingly, the performance drops below 3 MRows/s if systems of size N are solved. Thus, for a tile size of N, one pentadiagonal system is solved, and for a tile size of 64, $N/64$ pentadiagonal systems are solved with one function call. The ILU(0) factorization, which can be used to solve pentadiagonal systems exactly, performs poorly, if all bands are full with non-zero entries, and thus, the solving step is calculated with less parallelism.

For pentadiagonal systems, we obtain the amount of bytes read by `SgpsvInterleavedBatch` for $N = 2^{20}$ from NVIDIA Nsight Compute to

tile size	512	64	8
bytes read [MB]	46.149	45.528	32.327

although the size of the 5 bands and right-hand side is equal to 25.166 MB. The reduction and substitution kernel of `tridigpu` read 51 MB on the first stage, which is more than twice of the pentadiagonal system, due to reading the coarse solution in the substitution kernel. Nevertheless, the `tridigpu` banded solve achieves the same performance as the interleaved batched solve from cuSPARSE with a tile size of 64 for large N. As expected the equation throughput decreases if the number of bands are increased for `tridigpu` due to the increased amount of computational effort, which e.g. is required for the block multiplications.

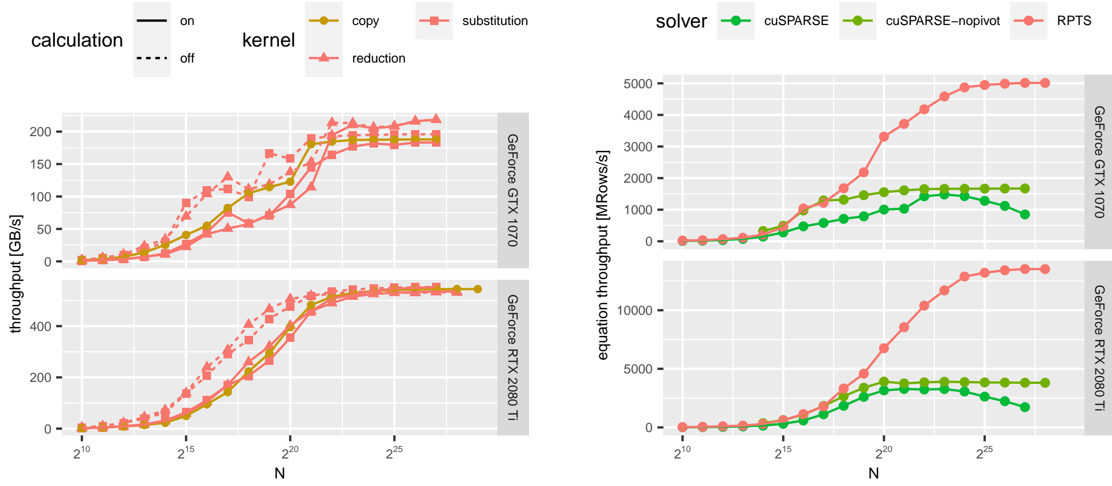


Figure 12.: Single-precision performance of scalar tridiagonal solvers for matrix 1 from Table 4 with size N. Left: RPTS global memory throughput of the finest stage. Right: performance comparison to the tridiagonal solver from cuSPARSE. Missing data points indicate that the required memory exceeds the available GPU memory.

2.7.3. Calculation of $A^{-1}\mathcal{D}$ with Sparse Right-Hand Sides and Solutions

We solve a scalar tridiagonal system $A\mathcal{X} = \mathcal{D}$ with many right-hand sides $\mathcal{D}$ with function `gtsv_csc2csc`. The results for `gtsv_csc2dense` with a sparse right-hand side and a dense solution are shown in Section 2.7.2. For $\mathcal{D}$, we take the transposed matrices of the Sparse Matrix Collection, which are listed in Table 17, and set the CSC column pointers of the result $\mathcal{X}$ to the same of $\mathcal{D}$. Note, that the matrices from the Sparse Matrix Collection are only used as right-hand sides, and are not inverted. With different sizes and number of non-zero elements they provide a test set for analyzing the function's performance depending on different sparsity patterns of the right-hand sides. The coefficients of the tridiagonal matrix A are generated with a uniform distribution between minus one and one and the number of right-hand sides, which are processed in one batch, is set to

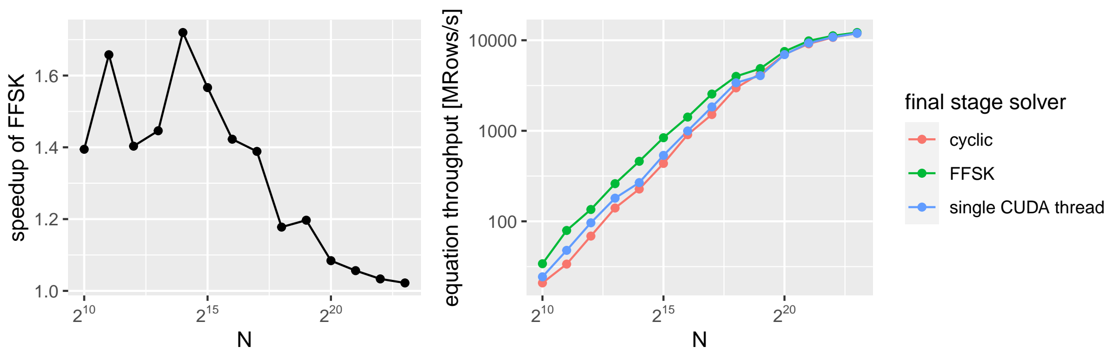


Figure 13.: Single precison scalar tridiagonal solver performance for different RPTS final stage solvers. Left: the corresponding speedup of FFSK relative to a final stage solver with a single CUDA thread.

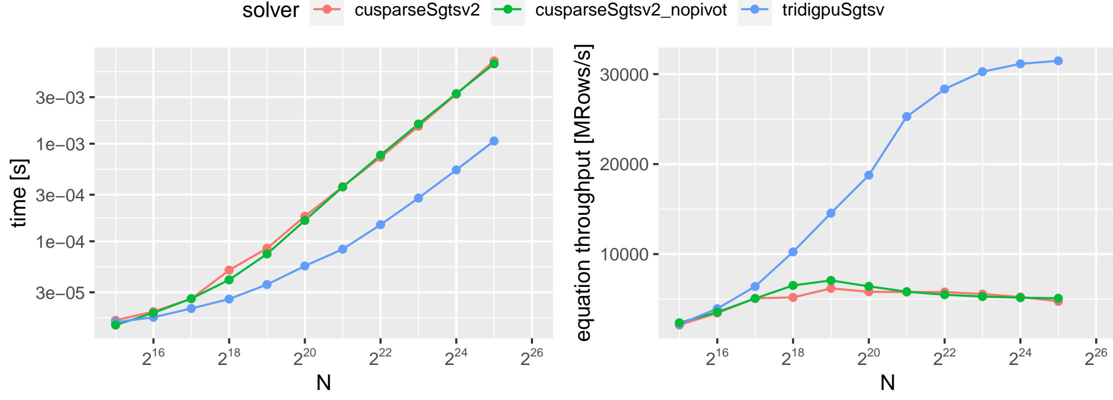


Figure 14.: Single precison scalar tridiagonal solver performance with $n_r = 32$ right-hand sides and N unknowns.

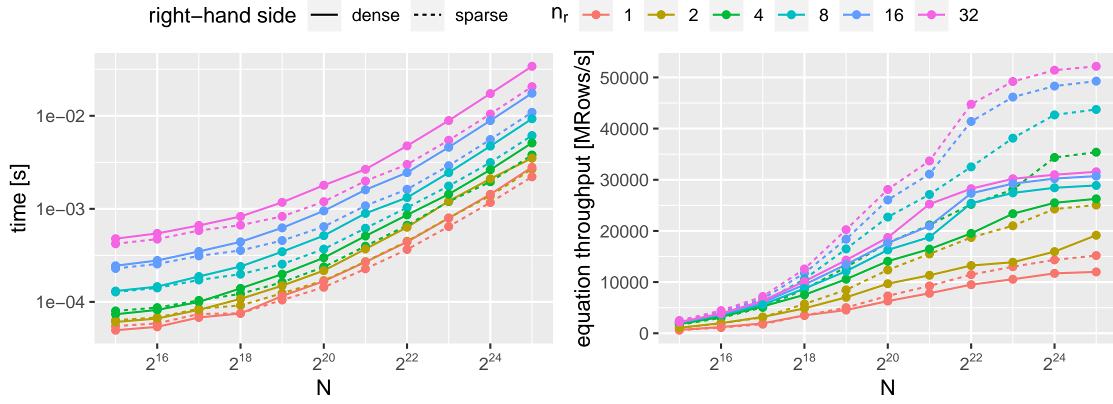


Figure 15.: Single precison scalar tridiagonal solver performance of `tridigpuSgtsv` and `tridigpuSgtsv_csc2dense` with varying number of right-hand sides and N unknowns.

Name	**Problem**	**DOFs**	**nnz**	`max_element`$(\texttt{nnz}_i)$
ECOLOGY2	2D/3D	999 999	4 995 991	5
THERMAL2	thermal	1 228 045	8 580 313	11
GEO_1438	structural	1 437 960	63 156 690	57
ATMOSMODL	fluid dynamics	1 489 752	10 319 760	7
HOOK_1498	structural	1 498 023	60 917 445	93
ML_GEER	structural	1 504 002	110 879 972	74
AF_SHELL10	structural	1 508 065	52 672 325	35
G3_CIRCUIT	circuit simulation	1 585 478	7 660 826	6
TRANSPORT	structural	1 602 111	23 500 731	15

Table 10.: Matrices from the Sparse Matrix Collection [27] which are used as sparse right-hand sides $\mathcal{D}$. `max_element`$(\texttt{nnz}_i)$ is the maximum number of non-zeros per row.

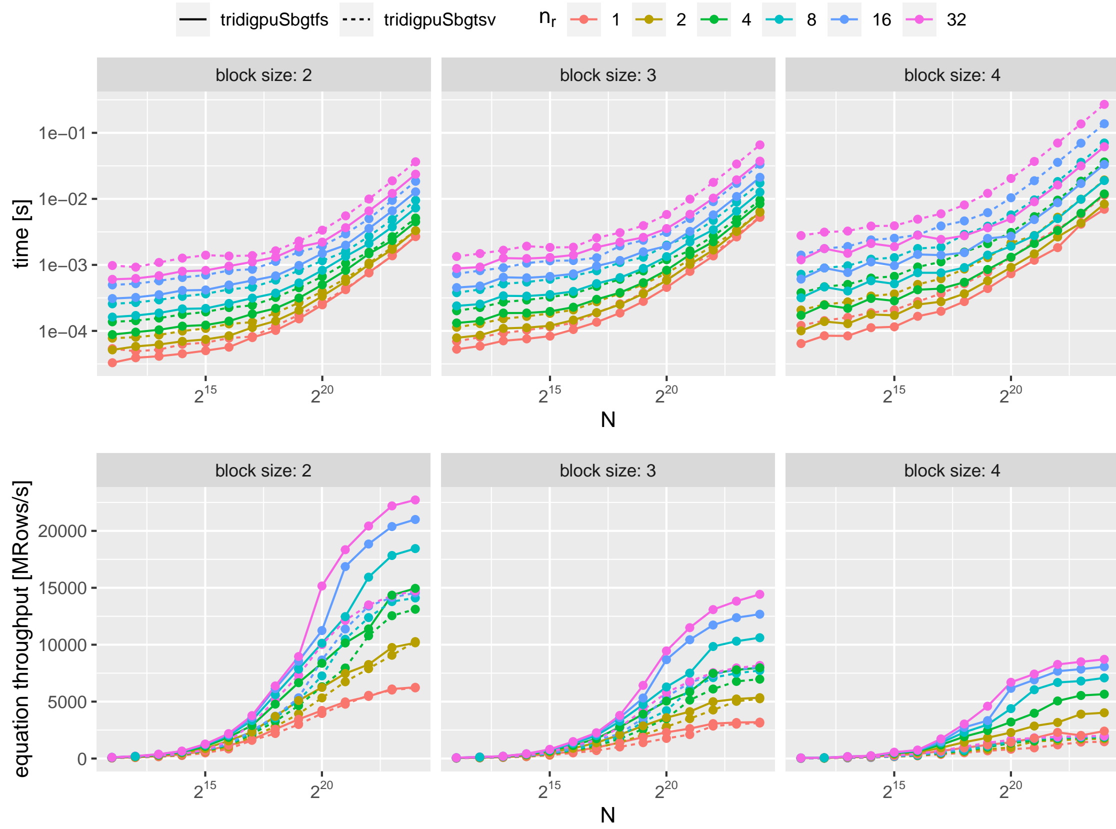


Figure 16.: Single precison block tridiagonal solver performance of `tridigpu` with varying number of right-hand sides and N unknowns. `bgtsv` repeatedly calculates the diagonalization, whereas `bgtfs` uses a given factorization of the tridiagonal system for the solution.

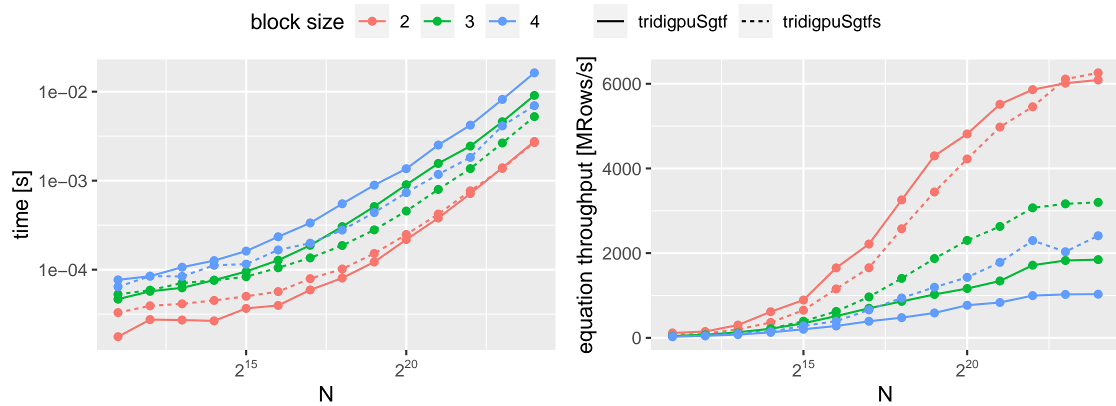


Figure 17.: Single precison block tridiagonal factorization performance (`tridigpuSgtf`) in comparison to the solving step (`tridigpuSgtfs`, $n_r = 1$).

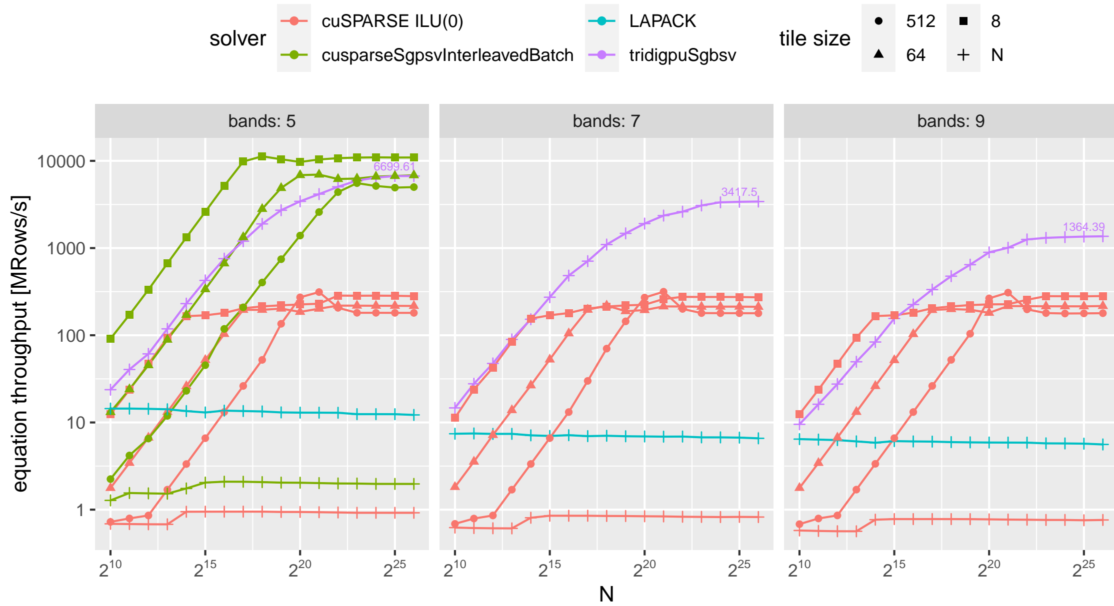


Figure 18.: Single precision banded solver performance. For `tridigpuSgbsv`, the maximum equation throughputs are shown.

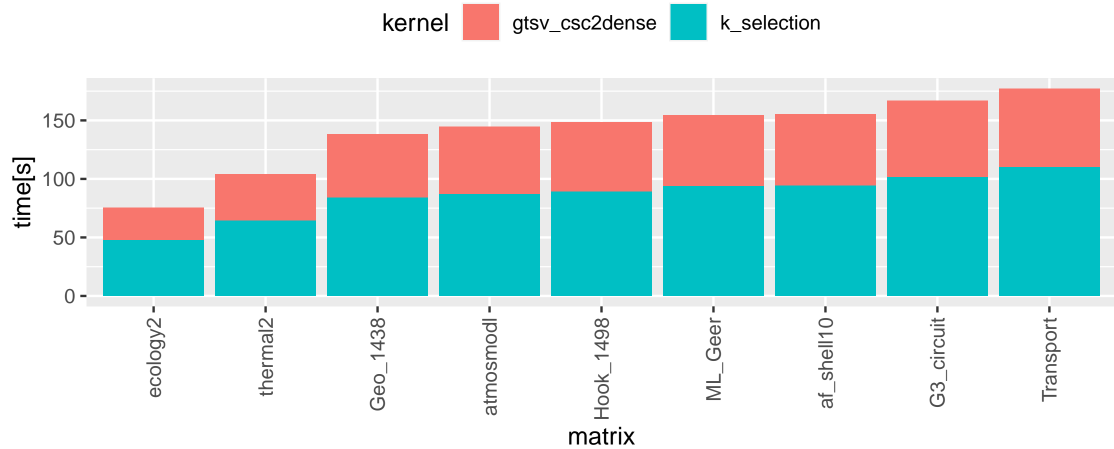


Figure 19.: `gtsv_csc2csc` runtime in single precision for many sparse right-hand sides $\mathcal{D}$ from the Sparse Matrix Collection (Section A.1.4).

$n'_r = 128$. The corresponding runtimes can be seen in Figure 19. More than half of the runtime is consumed by the selection of the largest absolute values of the dense tridiagonal solver result. For 10^6 single precision elements, the histogram256 calculating kernel of the k-selection algorithm runs with 320 GB/s on the RTX 2080 Ti, and is limited by the inter-warp aggregated irregular atomic additions in shared memory. In comparison to CUB's device histogram [84], that is a speedup of 1.57. The overall runtime is in order of hundreds of seconds for the chosen matrices, which is caused by the runtime complexity of $\mathcal{O}(N^2)$ for square matrices $\mathcal{D}$, as the hierarchical solve of one tridiagonal system is in $\mathcal{O}(N)$. A faster alternative would be to drop coefficients early in $\mathcal{D}$ during the solve, similar to how ILU drops coefficients early in A during a factorization.

However, this introduces more imprecisions and here we focus on the accuracy of the result. If A contains many independent small tridiagonal systems other algorithms, which dynamically solve only the tridiagonal systems with non-zero right-hand sides, are more efficient.

2.8. Summary and Outlook

In this section we discuss the limitations and future work for the presented function classes.

The scalar tridiagonal solve for multiple right-hand sides (`[c]gtsv`) is only limited by GPU memory bandwidth if the GPU has a sufficient amount of computational units for the corresponding precision. These functions run at hardware limit and cannot be further optimized.

The block tridiagonal solve and factorization are limited by thread execution latencies. In comparison to the scalar solves, the block tridiagonal solves have an increased computational effort, which cannot be hidden behind global memory accesses any more if only one warp is solving the inner partitions. For the future, this motivates us to increase the amount of parallelism in the matrix-matrix or matrix-vector operations during the solving of the inner partitions. However, this is not just a change in the parameterization, a significantly different parallelization strategy would be required. This new parallelization should overcome the limitation of $n \leq 4$ to some extent, maybe up to $n \leq 16$. But if we wanted to process even larger n efficiently, then yet another parallel strategy would have to be invented.

The sparse tridiagonal solves are efficient for sparse right-hand side input vectors, which fit into GPU memory cache because each CUDA block reads all right-hand side elements. Approximately 60% of the runtime of `gtsv_csc2csc` is limited by the performance of the k-selection algorithm. Therefore, increasing the performance of the device histogram by stronger data privatization or merging the k-selection into the substitution kernel of the scalar tridiagonal solver will be investigated.

To ensure performance portability across different GPU architectures, an auto-tuning mechanism to determine the optimal kernel parameters like the block dimension or partition size M would be useful in the future.

2.9. Conclusion

Our `tridigpu` CUDA library for block cyclic, tridiagonal, and banded systems covers multiple problem classes which often appear in practice. We have discussed the C API, the underlying algorithms, the numerical stability, and the performance of these problem

classes in detail. In comparison, we showed that our solvers outperform state-of-the-art tridiagonal solvers as they are implemented in cuSPARSE, and that our banded solver keeps up with batched pentadiagonal solvers, although it is able to solve very large pentadiagonal systems and not only batched systems.

3. Sparse Matrix-Vector Multiplication with Segmented Reduction

Abstract CSR matrix-vector products (SpMV) on GPUs often use specialized implementations for e.g. very long or short matrix rows and depend also on built-in atomic functions. In this work, we present a CUDA SpMV implementation by using segmented reduction algorithms as a building block but without any specialization with respect to the sparsity pattern of the matrix. We provide three schemes which differ in their work balancing technique between the CUDA blocks. Two of which are free of atomic operations and thus support general types and operators for computing a generalized SpMV product in e.g. graph applications. Despite the generality, our implementation competes with the performance of standard SpMV with cuSPARSE, which we show for an RTX 2080 Ti and large single precision matrices from the Sparse Matrix Collection.

3.1. Introduction

The sparse matrix-vector product (SpMV) represents an important building block for many scientific applications and problems like sparse iterative solvers [104], the power iteration for largest eigenvalue calculation [28], Google's page rank algorithm [117], or graph algorithms expressed in terms of sparse linear algebra [82].

The strong irregularities of sparse matrices with respect to the location of the non-zero elements represents a major difficulty when implementing an efficient SpMV product on a GPU, which requires fine grained parallelism and work balancing among the threads. If for example one GPU thread processes one single sparse matrix row to calculate one result y_i of the product $y = Ax$, some threads might process much longer matrix rows, which will result in thread divergence, uncoalesced global memory loads, and threads stalling as they are waiting for other threads in the CUDA block. The latter approach was evaluated for the CSR format by Bell and Garland [11, 12] and denoted with CSR-scalar. Additionally, they proposed a CSR-vector kernel, which assigns a group of threads with a globally fixed size to one matrix row. However, the problem of work imbalances between GPU threads remains if e.g. many matrix rows are shorter than the group size. Many approaches to fix these work imbalances by changing the sparse matrix format can be found in literature[1] but integrating these additional formats into an existing application always requires the potentially expensive format conversions. Therefore, we focus on the widely used Compressed Sparse Row (CSR) format in this chapter and do not require any modifications to the input matrix A.

For a matrix A with N columns, our CUDA SpMV implementation (SRCSR) includes the solution of the generalized SpMV problem

$$y_i = \beta \otimes y_i + \alpha \otimes (A_{i,0} \otimes x_0 \oplus A_{i,1} \otimes x_1 \oplus \cdots \oplus A_{i,N-1} \otimes x_{N-1}), \tag{19}$$

[1] For a summary, see [46].

with abstract multiplication ⊗ and addition ⊕ operators. SRCSR uses a segmented reduction in on-chip memory, with each matrix row representing one segment, to compute the sum of the products $A_{i,j} \otimes x_j$. With a preprocessing consuming between 10% and 20% of the actual SpMV runtime, we assign the non-zero coefficients of A evenly (SRCSR-balanced) or as evenly as possible (SRCSR-adaptive) to the CUDA blocks. For SRCSR-adaptive, each matrix row is assigned to one CUDA block but one CUDA block might be assigned to multiple matrix rows, whereas SRCSR-balanced might assign multiple CUDA blocks to the same matrix row. For both schemes, work imbalances occur between the CUDA blocks in case of many matrix rows without any non-zero coefficient because the SRCSR SpMV kernels also write the result (which is equal to zero) for these rows. Additionally for SRCSR-adaptive, work imbalances between CUDA blocks might decrease efficiency due to tail-effects as a block working on a very long matrix row is still running while all other blocks already finished.

By using the segmented reduction as a building block for SpMV, we leverage highly optimized segmented reduction codes and simultaneously provide a high level of generality in the implementation with the support for arbitrary types and operators. Often SpMV implementations for GPUs do not support arbitrary types because partial results for the same matrix row are accumulated by built-in atomic instructions [47, 80], which further requires a previous zero initialization of the output vector. SRCSR-adaptive does not use any atomic instructions, whereas we provide two different ways for SRCSR-balanced for the accumulation of the partial results: first, by using built-in atomic instructions (SRCSR-balanced-atomic), and second, by launching a post processing kernel (SRCSR-balanced-postprocessing), which accumulates the partial results with another segmented reduction. SRCSR-balanced-atomic avoids an additional kernel launch by atomics at the cost of losing support for general types, and SRCSR-balanced-postprocessing supports general types at the cost of an additional kernel launch.

The rest of the chapter continues with discussing related work in Section 3.2, presenting an overview of the SRCSR algorithm in Section 3.4, giving further details on the implementation in Section 3.5, and showing the performance results in Section 3.6.

3.2. Related Work

Related to our contributions are other SpMV implementations which do not require any modification of the input CSR matrix and improve work balancing among threads in comparison to the static approaches CSR-scalar and CSR-vector by Bell and Garland [11, 12].

Liu and Schmidt [80] proposed LightSpMV which is based on CSR-vector but with a dynamic matrix row assignment to the groups of threads. After each processed matrix row, a group of threads atomically retrieves the next row index for processing. In comparison to CSR-vector, this prevents low utilization if e.g. only one group of threads in each CUDA block processes one long matrix row because the other groups of threads dynamically retrieve another row for processing and are not idling. However, the tail-effect (see Section 3.1) for LightSpMV is even worse than for SRCSR-adaptive because only one group of threads is operating on one potentially long matrix row, whereas for SRCSR-adaptive that is one CUDA block. Moreover, LightSpMV uses atomic operations to accumulate the partial sums for one matrix row and is thus limited in its type flexibility.

Ashari et al. [9] propose ACSR which bins matrix rows depending on their length and launches different kernels for different bins with dynamic parallism. The preprocessing

cost is relatively high and consumes more than two SpMV operations[2]. Moreover, the input matrix is not modified, which results in strided global memory accesses if short and long rows occur in an alternating fashion.

Greathouse and Daga [56] propose CSR-Stream and CSR-Adaptive. The OpenCL implementation of CSR-Stream is similar to our SRCSR-balanced scheme uses a preprocessing step to assign matrix rows dynamically wavefronts and loads the non-zero matrix coefficients into on-chip memory. But contrary to our SRCSR schemes, CSR-Stream uses one GPU thread to calculate the reduction along a matrix row in on-chip memory, which limits the CSR-Stream implementation to a certain row length. For arbitrary row lengths, they provide CSR-Adaptive, which dynamically uses either CSR-Stream or a classical CSR-Vector implementation to process the matrix row.

Flegar und Quintana-Ortí [47] propose a work balanced SpMV implementation (CSR-I) which assigns the non-zero elements evenly among the threads. The threads read the matrix coefficients coalesced from global memory, multiply the the vector coefficient, and update their local accumulator. In case a thread encounters a coefficient from a new matrix row, the result is written atomically warp-aggregated into global memory. This requires the zero initialization of the output vector and limits possible types to the atomically supported ones.

A SpMV implementation with the support for general types is available in CUB [85], which also considers empty matrix rows when distributing the workload among the threads. However, our SRCSR SpMV implementations provide a more fine-grained type configuration (configurable types for matrix coeffcients, intermediate accumulator type, result type, and output type) and support for user defined abstract addition and multiplication operators.

3.3. Contributions

We provide the CUDA implementation of a generalized sparse CSR matrix-vector product with the support for fine-grained user types and abstract multiplication $\otimes$ and addition $\oplus$ operators by two major schemes, which use segmented reduction algorithms as internal building block:

SRCSR-adaptive :

- nearly work balanced with respect to non-zero coefficients of A
- CUDA block exclusive matrix row assignment (potential performance decreasing tail-effect)
- atomic-free
- support for abstract user types and operators
- worst for matrices with only a few very long rows (low occupancy)

SRCSR-balanced :

- work balanced with respect to non-zero coefficients of A
- multiple CUDA blocks operate on the same matrix row

-atomic :

  - atomics for partial sum accumulation between different CUDA blocks

[2]implicitly derived from author statement in Section I: 'less than 3 SpMV operations'

- type and operator support limited by atomic built-in functions
- worse for matrices with only a few very long rows (atomic collisions between CUDA blocks)

-postprocess :

- atomic-free
- support for abstract user types and operators
- additional post processing kernel to accumulate partial sums

Besides the classical SpMV operation $y = \beta y + \alpha Ax$, many other applications can use our SpMV implementation because of its generality, e.g. GraphBLAS applications [82] or linear forest computations (see Chapter 4).

3.4. Algorithms

Algorithm 4: Computing the start rows for SRCSR-balanced. N is the number of matrix rows and w is the amount of non-zero matrix coefficients, which is processed by one CUDA block.

```
   Data: CSR row_ptrs, w
   Result: row_block_start
 1 ∀ b ∈ [0, num_blocks] :  row_block_start[b] = −1
 2 row_block_start[0] = 0
 3 for i = 0, ..., N − 1 do in parallel
 4 |   j = row_ptrs[i + 1] − 1
 5 |   k = row_ptrs[i] − 1
 6 |   l = j/w
 7 |   m = k/w
 8 |   if l ≠ m then
 9 |   |   row_block_start[m + 1] = i
10 |   end
11 end
12 row_block_start[num_blocks] = N
13 inclusive_max_scan(row_block_start)
```

3.4.1. Preprocessing

For SRCSR-balanced, an equal amount of non-zero matrix coefficients is assigned to each CUDA block, which gives the first non-zero element of a CUDA block as the number of non-zero elements per CUDA block multiplied with the block ID. The locations of the results in vector y for a general SpMV operation (Eq. 19) are computed by a preprocessing kernel, which returns the start and end matrix row for each CUDA block (Algorithm 4). The input of Algorithm 4 is the `row_ptrs` vector which saves the index in CSR not-null space for each matrix row. The output is vector `row_block_start`, which contains for each CUDA block b the matrix row range [`row_block_start[`b`]`, `row_block_start[`$b+1$`]`). In a parallel implementation, Algorithm 4 is a scatter operation with the complexity increasing linearly with the number of matrix rows ($\mathcal{O}(N)$) and thus its runtime is only a fraction

Algorithm 5: Computing the start rows for SRCSR-adaptive. N is the number of matrix rows and w is the target amount of non-zero matrix coefficients, which is processed by one CUDA block.

Data: CSR `row_ptrs`, w
Result: `row_block_start`

1 $\forall\, b \in [0, \texttt{max_num_blocks}]: \ \texttt{row_block_start}[b] = -1$
2 $\texttt{row_block_start}[0] = 0$
3 **for** $i = 0, \ldots, N-1$ **do in parallel**
4 $\quad j = \texttt{row_ptrs}[i+1]$
5 $\quad k = \texttt{row_ptrs}[i]$
6 $\quad l = j/w$
7 $\quad m = k/w$
8 $\quad$ **if** $l \neq m \wedge l \neq 0 \wedge l \neq \texttt{max_num_blocks} \wedge j \neq k$ **then**
9 $\quad\quad \texttt{row_block_start}[l] = i+1$
10 $\quad$ **end**
11 **end**
12 `row_block_start[max_num_blocks]` $= N$
13 `num_blocks = select_if_unequal_minus_one(row_block_start, row_block_start) - 1`

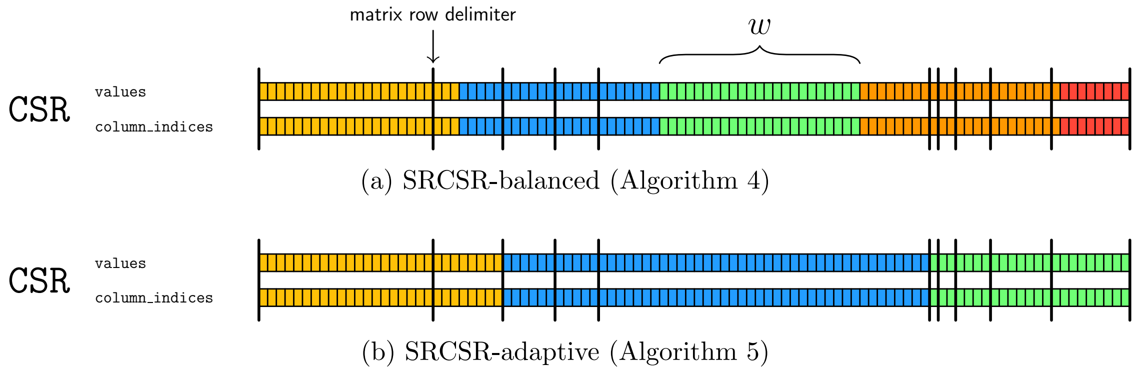


Figure 20.: Small example for matrix row assignment to CUDA blocks in CSR not-null space with $w = 23$. The colors represent different CUDA blocks.

of the actual SpMV computation. The inclusive max scan in Line 13 fills the unwritten values in `row_block_start` in case multiple CUDA blocks operate on the same matrix row.

Analogously, Algorithm 5 shows the preprocessing for SRCSR-adaptive. A priori the number of CUDA blocks for the SRCSR-adaptive kernel is unknown but has an upper limit of $\texttt{nnz}/w + 1$, where w represents the target amount of non-zero matrix coefficients per CUDA block. In fact, some CUDA blocks might process less or more than w non-zero coefficients. Therefore, the actual number of CUDA blocks is computed in Line 13 where all written elements in `row_block_start` are selected and counted, which is returned by the function `select_if_unequal_minus_one()`.

Figure 20 visualizes the matrix row assignment to CUDA blocks of Algorithm 4 & 5 for a small example CSR matrix.

Alternatively to a separate preprocessing kernel, the matrix rows could be obtained with a binary search in the beginning of the SpMV kernel.

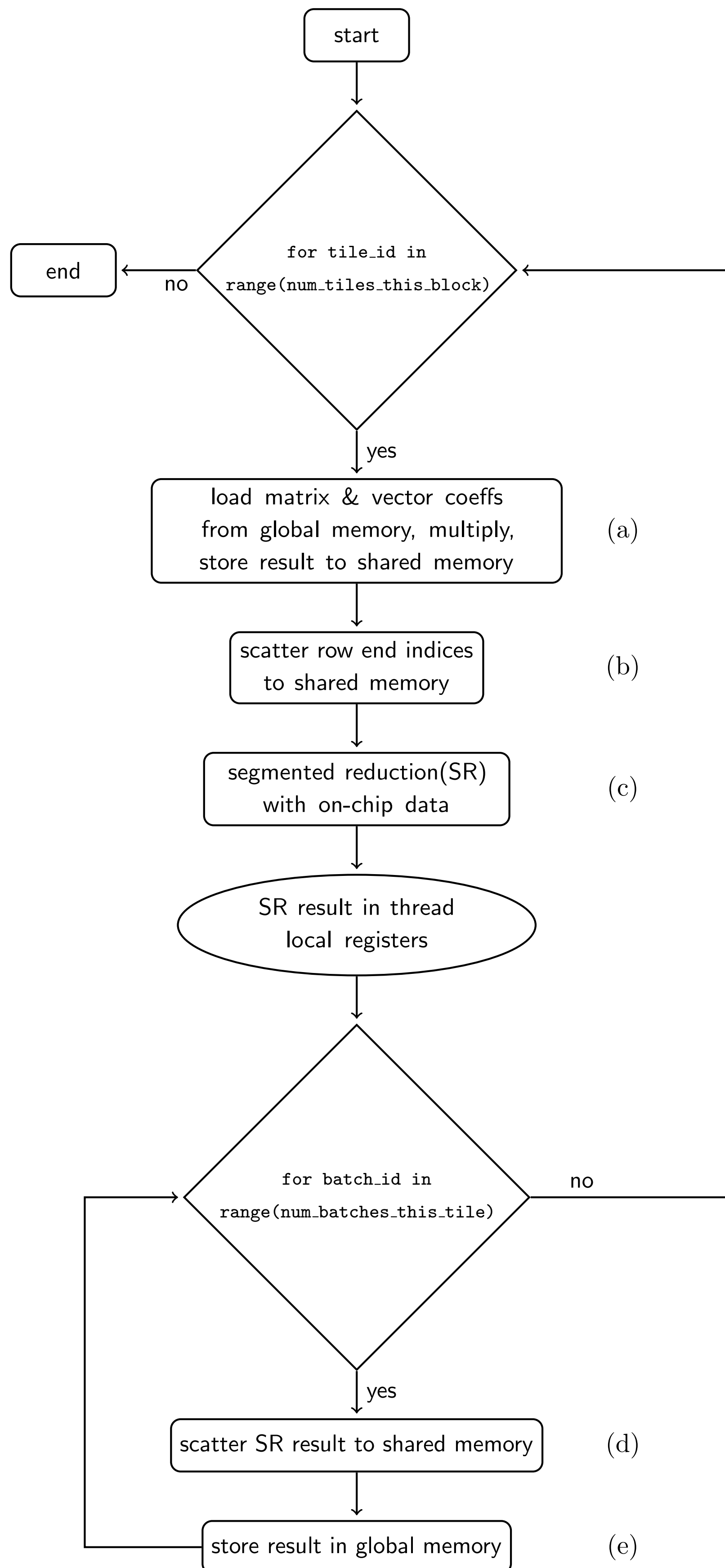


Figure 21.: SRCSR flow chart for each CUDA block (SR = segmented reduction).

3.4.2. SRCSR

In our segmented reduction based CSR matrix-vector product, each CUDA block is assigned to a certain number of non-zero matrix coefficients and matrix rows. In general, all SRCSR schemes load the matrix coefficients and column indices coalesced and the vector coefficients irregularly depending on the column indices to on-chip memory (shared memory). Afterward, the (partial) result is computed and stored back to global memory in coalesced fashion.

A simplified control flow is given by Figure 21. Steps (a)-(e) are executed in a for-loop over *tiles* because the amount of data, which is assigned to a CUDA block (parameter w), might exceed the limit of available on-chip memory and the segmented reduction (Step (c)) operates only with on-chip data. For matrix rows exceeding the tile borders, the partial result of the previous tile is added to the first element of the next tile. In Step (b), the threads scatter the row indices of the tile into their position in the partial CSR not-null space located in shared memory. The row end indices serve as flags for the segmented reduction in Step (c) and also contain the information of the result location in vector y, which is used in Steps (d) & (e). After the segmented reduction, the results are either loaded from shared memory to thread local registers, or are already placed in registers depending on the type of segmented reduction algorithm. Consequently, the shared memory is reused in a for-loop over *batches* to store the results coalesced in vector y. The necessity of this second for-loop stems from the arbitrary amount of empty matrix rows, which are part of the current tile.

3.5. Implementation

The implementation is hosted on the GitLab-Server[3] of the Application Specific Computing group at the Heidelberg University and published under a 3-Clause BSD license.

3.5.1. Segmented Reduction Algorithms

The implementation of SRCSR supports two major algorithms to compute the segmented reduction in Step (c), Figure 21: the block primitive `BlockScan` of CUB [84] or our own warp transposed segmented reduction algorithm (Section 3.5.2). CUB's block primitive provides three different algorithms: raking, raking-memoize, warp-scans.

3.5.2. Warp Transposed Segmented Reduction

In Section 2.6.3, we showed the data transposition in shared memory for coalesced loading of the tridiagonal system from global memory and subsequently the solving of 32 partitions with one CUDA warp where one thread operates sequentially on its own partition. We used the same idea to implement a segmented reduction with one CUDA warp. Each thread is assigned to partition of the input data for the segmented reduction, which is shown in Figure 22 by one row for one thread. Keep in mind that this is just a visual 2D representation of the CSR data. In fact, the CSR data is located continuously in shared memory. In the first Step (1), each thread computes its segmented reduction on its own partition of the data. Subsequently in Step (2), a segmented scan with warp shuffle functions is used to exchange the potentially required correction (Step (3)) for the first segment of each thread. Note, that in the example, thread 2 has not any segment delimiter and is thus not correcting any output value. This algorithm is naturally free of shared

[3] `https://mp-force.ziti.uni-heidelberg.de/cklein/srcsrmv`

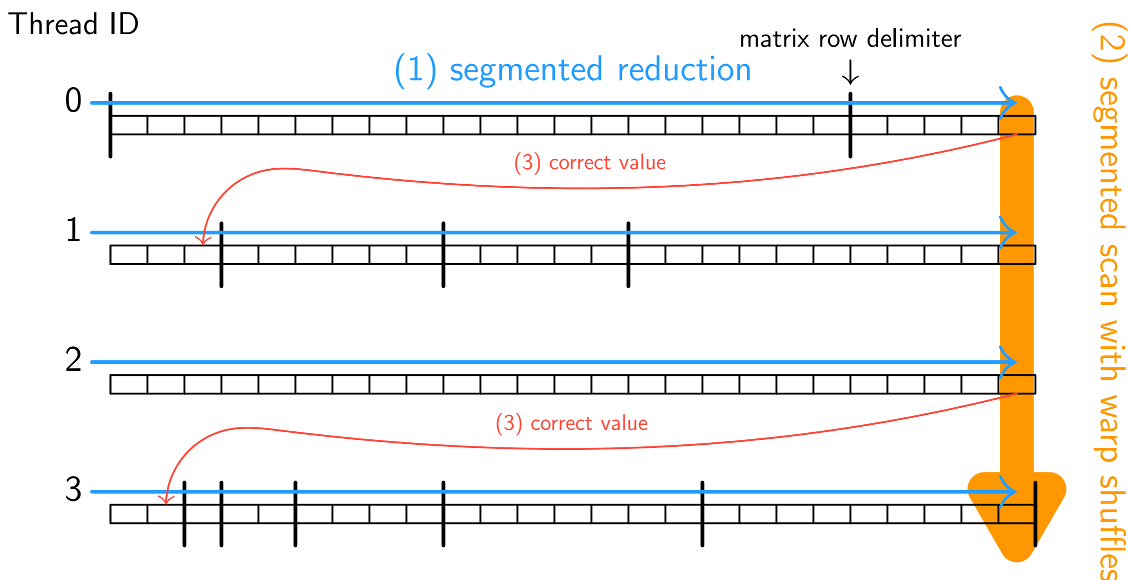


Figure 22.: Warp transposed segmented reduction in three steps on small example CSR matrix from Figure 20 (warp size = 4).

memory bank conflicts if the partition size is odd. For even partition sizes, a padding of the partitions in shared memory ensures zero bank conflicts.

3.5.3. Use of Atomic Operations

As shown by the contributions in Section 3.3 only the SRCSR-balanced-atomic scheme makes use of atomic operations to accumulate the partial sums of the matrix rows. This requires the zero initialization of the output vector y but instead of completely initializing y, we only initialize the few values in y, which are written atomically. A CUDA block only uses the atomic operations for matrix rows, which are shared with other blocks, e.g. the blue block in Figure 20a, contributes to four matrix rows but only executes two atomic additions.

3.5.4. Type Configuration

To maximize efficiency of a generalized SpMV with more complex user defined types and operators, a fine grained type configuration is required for the SpMV kernels.

Figure 23 shows the type conversion flow during a generalized SpMV computation. The matrix coefficients and vector elements are loaded from global memory via general iterator types, combined with abstract multiplication operator $\otimes$, and saved in shared memory. The access to shared memory can be controlled with a separate iterator type. This can be used for performance improvements if the output type of $\otimes$ is a large `struct` with multiple member variables. In that case, the abstract iterator provides a struct of arrays (SoA) access while giving an array of structs (AoS) view to the programmer.

If SRCSR is used with CUB's segmented reduction (SR) algorithm, the output type of $\otimes$, is subsequently converted to the in- and output type of the SR while it is loaded to thread local registers. Note, that the requirement of commutativity for the input type by highly parallel (segmented) reduction/scan algorithms forces the same type for in- and output. This is different for sequential reduction/scan algorithms where the accumulator type can be different to the input type. For SRCSR, the type conversion from the output

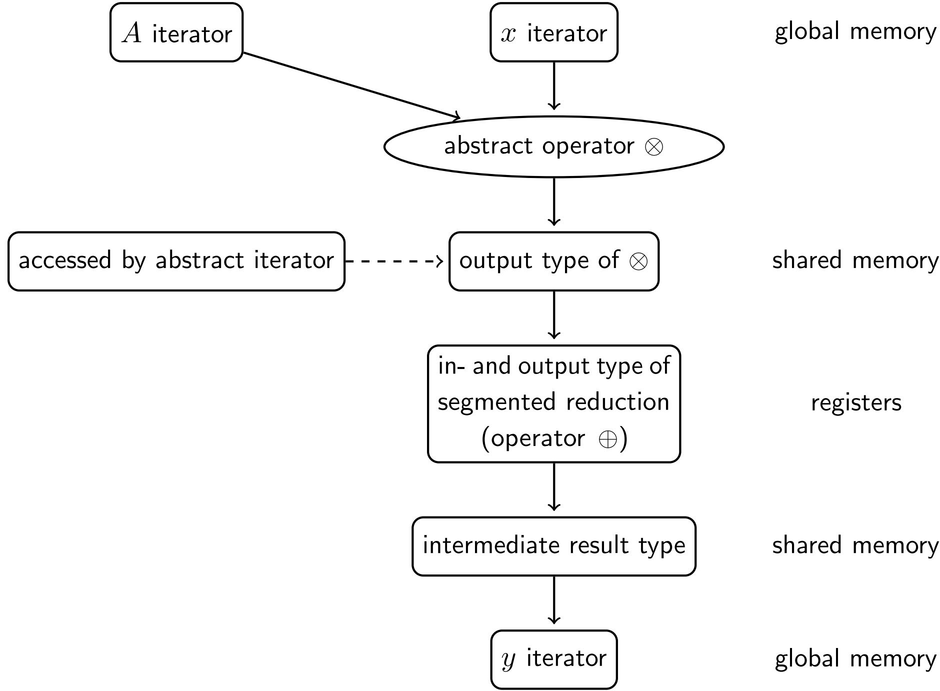


Figure 23.: SRCSR type configuration for generalized sparse matrix-vector products in case of CUB's segmented reduction algorithm. The right column denotes typical data locations in GPU memory and the arrows indicate the direction of type conversion. Each box represents a user configurable type.

type of $\otimes$ to the SR input type can be used to significantly reduce the size of required shared memory size per CUDA block. An example is a segmented reduction to find the two maximum values in each segment. The value type for the reduction would be a pair of two values because value and accumulator type must be the same. Therefore, each input value would be converted to a pair consisting of the value itself and a placeholder. The pair has twice the size of the input value type and would not be saved in shared memory.

After the SR with a potentially large type, the SR result is converted to an intermediate result type, which might be smaller. In the end, the intermediate result type is assigned to an iterator on output vector y.

If the warp transposed segmented reduction from Section 3.5.2 is used, the output type of $\otimes$ must be the same as the input type for the SR.

3.5.5. Abstract Operators

The SRCSR kernels are parametrized by three lambda functions, which implement the abstract operators $\otimes$ & $\oplus$ and control how the output is written to vector y. In the first lambda, the matrix coefficients from A and the vector elements from x are loaded and multiplied ($\otimes$). The second lambda represents the binary operator ($\oplus$) for the segmented reduction, and the third lambda controls how the result is written to global memory, e.g. the computation of $y = Ax$, $y = \beta y + Ax$, $y = z + y + Ax$, or even $y = z + y + TAx$ with a diagonal matrix T.

For the lambda which implements operator $\otimes$ two type signatures are supported:

- `operator_otimes(int nz_id)`, and

- `operator_otimes(int nz_id, int row_id)`,

where `nz_id=0,...,nnz-1` represents the non-zero index in CSR not null space, which is used to index the CSR `values` and `col_indices` ranges. The first signature represents the default and for a standard SpMV we would define:

```
auto operator_otimes = [&] __device__ (int nz_id) {
  return A_values[nz_id] * x[A_col_indices[nz_id]];
};
```

The second signature is special, as it allows the usage of the row index for each non-zero matrix coefficient. A synthetic example is the SpMV computation where some rows only contribute partially to the SpMV result:

```
auto operator_otimes = [&] __device__ (int nz_id, int row_id) {
  const bool b0 = predicate[row_id];
  const bool b1 = nz_id % 2 == 0;
  const auto t = A_values[nz_id] * x[A_col_indices[nz_id]];
  return b0 || b1 ? t : 0;
};
```

From a user perspective it seems as if the matrix has a COO format but the row indices are computed with minimal overhead in on-chip memory without any additional memory requirements. With type traits, it is determined at compile time if the row indices must be computed or not. Generating the row indices on-chip saves time consuming CSR to COO conversions and avoids the usage of additional memory to save the row indices. This feature is used in Chapter 4 for a linear forest computation.

3.5.6. Auto-Tuning

The SRCSR kernels have four tuning parameters which affect their runtime on a GPU architecture:

- the CUDA block dimension,
- the amount of non-zero coefficients, which is processed in one tile (see Figure 21),
- the target number of non-zero coefficients per CUDA block w (see Figure 20), and
- the segmented reduction algorithm (see Section 3.5.1).

With only four options for each parameter, this multiplies to 256 different kernel configurations. In addition, the optimal parameter set might differ between GPUs of different architectures and different non-zero matrix patterns. Also, the optimal parameter set is different for standard and generalized SpMV due to fundamental (`float`, `double`) and abstract types, respectively. Therefore, the source code includes an auto-tuning mechanism to determine the optimal parameter set by benchmarking the cartesian product of reasonable parameter choices.

3.5.7. Partial Sparse Matrix-Vector Products

The SRCSR kernels support partial SpMV computations $d_i = \sum_j A_i j x_j$, with user defined limits $i = i_0, \ldots, i_1$. This can be used for parallel operations on colored graphs. For a

matrix	N	nnz
AF_SHELL8	504 855	17 588 875
ANISO1	6 250 000	56 220 004
ANISO2	6 250 000	56 220 004
ANISO3	6 250 000	56 220 004
ATMOSMODD	1 270 432	8 814 880
ATMOSMODJ	1 270 432	8 814 880
ATMOSMODL	1 489 752	10 319 760
ATMOSMODM	1 489 752	10 319 760
BUMP_2911	2 911 419	127 729 899
CUBE_COUP_DT0	2 164 760	127 206 144
CURLCURL_3	1 219 574	13 544 618
CURLCURL_4	2 380 515	26 515 867
ECOLOGY1	1 000 000	4 996 000
ECOLOGY2	999 999	4 995 991
G3_CIRCUIT	1 585 478	7 660 826
GEO_1438	1 437 960	63 156 690
HOOK_1498	1 498 023	60 917 445
LONG_COUP_DT0	1 470 152	87 088 992
ML_GEER	1 504 002	110 879 972
STOCF-1465	1 465 137	21 005 389
THERMAL2	1 228 045	8 580 313
TRANSPORT	1 602 111	23 500 731

Table 11.: Test matrices which are either from the Sparse Matrix Collection [27] or taken from [72] (ANISO{1,2,3}).

colored graph, neighboring vertices have a different color, thus a result for a vertex which depends on the neighbors can safely be computed and written to memory for one color in parallel. If matrix A is permuted such that the vertices of the same color have subsequent vertex indices, we can apply our generalized SpMV to vertices of the same color by setting the bounds i_0 and i_1 appropriately.

3.6. Results

For the results presented in this chapter, we use a machine with CentOS 7, CUDA Toolkit 11.4.48, CUDA driver 510.60.02, GCC 10.2.0, a GeForce RTX 2080 Ti. All performance measurements are done in single precision because the RTX 2080 Ti only has a few double precision units. For double precision performance, professional accelerator graphics cards can be used (A100, V100, etc.). The test matrices are listen in Table 11, which are mostly from the Sparse Matrix Collection [27].

3.6.1. Overall Performance

In this section, we compare our SRCSR kernels against the highly optimized SpMV kernels from cuSPARSE [90] and evaluate the time consumption of the preprocessing.

For that purpose, Figure 24 shows the relative time of the standard SpMV operation $y = \beta y + \alpha Ax$, with the slowest SpMV kernel always having a relative runtime of 1.0. The presented time is the average of 300 SpMV operations measured with two CUDA events. The SRCSR-balanced-postprocess scheme is most often the slowest scheme because it requires some kernel launches in the post-processing. Without preprocessing (Figure 24, top), SRCSR-adaptive and SRCSR-balanced-atomic compete with cuSPARSE and are

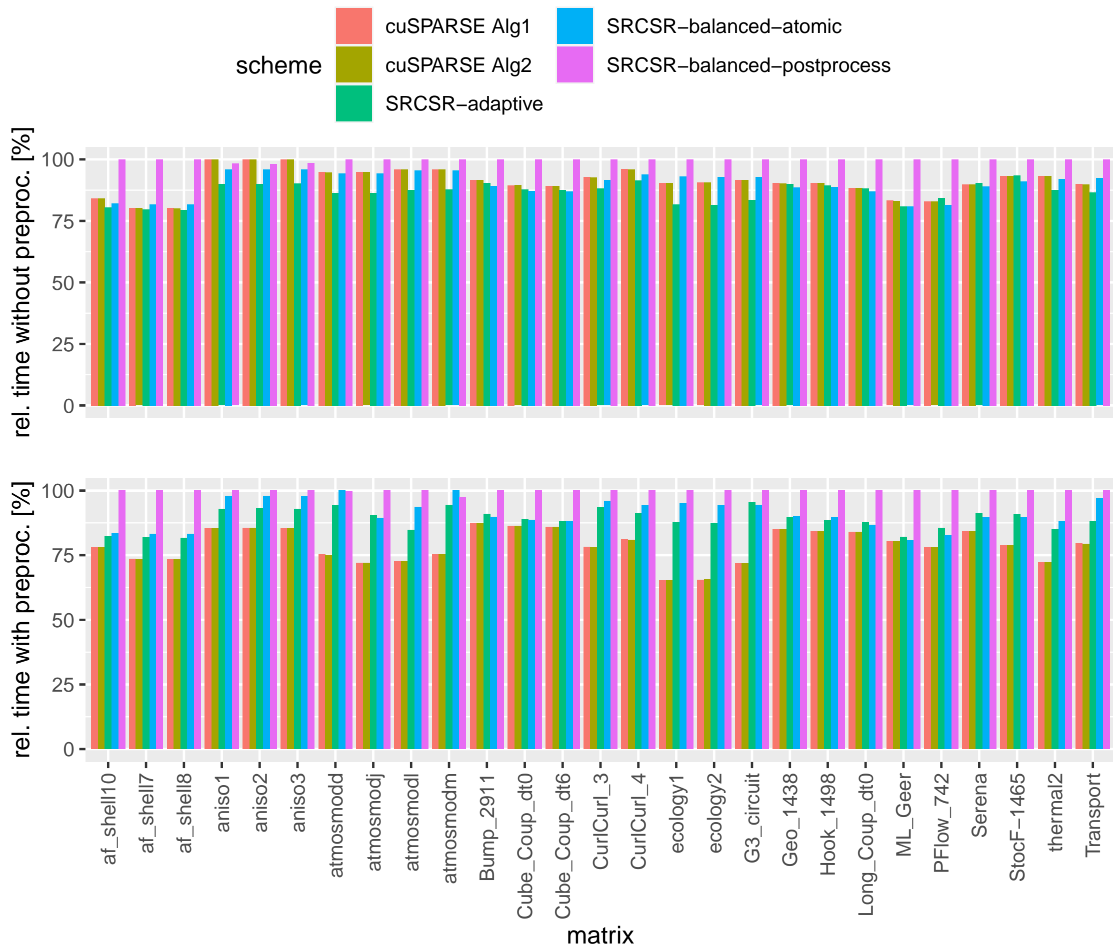


Figure 24.: SpMV performance comparison for operation $y = \beta y + \alpha Ax$ on single precision matrices from Table 11. Top: times without the SRCSR preprocessing. Bottom: with SRCSR preprocessing. The cuSPARSE times are equal in both plots.

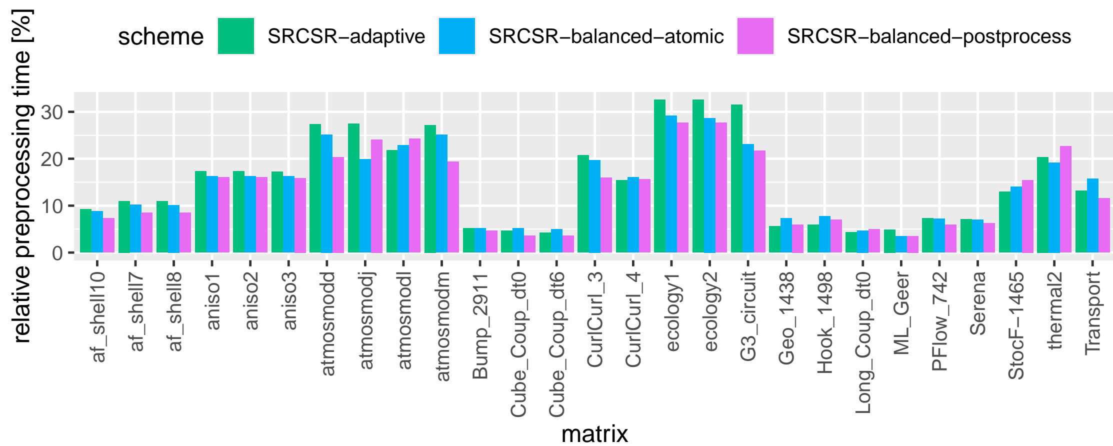


Figure 25.: Relative time for preprocessing of SRCSR SpMV for operation $y = \beta y + \alpha Ax$ on single precision matrices from Table 11. The time is relative to the SpMV + preprocessing time.

sometimes even faster. This benchmark is relevant for applications which execute SpMV often on the same sparsity pattern, such that the preprocessing is executed only once, and thus consumes a negligible time fraction of the overall runtime. The upper plot of Figure 24 shows the relative runtime with the SRCSR preprocessing, which unsurprisingly shows cuSPARSE most often as the fastest scheme. SRCSR has a much more general implementation than the assembly level optimized SpMV kernels of cuSPARSE, but still achieves a similar performance. Sometimes SRCSR-balanced-atomic performs worse than SRCSR-adaptive because of an additional kernel launch, which is required to compute the multiplication of y with β. SRCSR-adaptive does not require an additional kernel launch because the update on y is not made atomically.

In Figure 24, we showed the best performance results from our auto tuning framework (Section 3.5.6), but we want to point out that, that the SRCSR SpMV operation with our warp transposed segmented reduction from Section 3.5.2 performs as well as with CUB's segmented scan block primitive (Section 3.5.1).

The relative runtime of the preprocessing in comparison to the total runtime (SpMV + preprocessing) is shown in Figure 25. As expected, the preprocessing time is relatively longer (up to 30%) for matrices, which only have a few non-zero coefficients per row. For matrices with many non-zero coefficients per row (BUMP_2911, ML_GEER, GEO_1438, etc.), the preprocessing only takes up to 5% of the runtime.

3.6.2. General Applications of SRCSR

In Section 3.5.4, we pointed out the type generality of the SRCSR kernels, which we show for four applications in this section. An advanced application (linear forest extraction) is shown in Chapter 4. The four applications use the exact same SRCSR-adaptive kernel but with different type and operator instantiations, and are listed in increasing order of resource (registers, type sizes, operator complexity) consumption:

- **max1:** compute the maximum value of $\texttt{max}_j(A_{i,j})$ for each i.
- **max1-with-col:** compute the maximum value of $\texttt{max}_j(j \cdot A_{i,j})$ for each i.
- **arg-max1:** compute the index for the maximization of $\texttt{argmax}_j(A_{i,j})$ for each i.
- **arg-max2:** compute the indices j_0, j_1 with maximal $\texttt{argmax}_{j_0 \neq j_1}(A_{i,j_0})$ and $\texttt{argmax}_{j_1 \neq j_0}(A_{i,j_1})$ for each i.

We compare these applications with standard SpMV in Figure 26, which shows the time relative to the longest running scheme for a matrix. arg-max2 is the most resource consuming scheme, and consequently also has the longest runtimes. This is caused by the size of the input type for the segmented reduction. To find the columns of the two maximum values in each matrix row the input type must save the two single precision values (8 Byte) and the two column indices (8 Byte), which is in total 16 Byte. This is four times larger than the reduction input type for standard SpMV. Implementing arg-max2 with SRCSR is approximately one order of magnitude faster than an implementation with the segmented reduction algorithm from CUB [84], which is caused by the fine-grained type configuration of SRCSR (see Section 3.5.4).

3.7. Conclusion

In this chapter, we have shown a CUDA implementation for a generalized sparse matrix-vector multiplication, which uses segmented reduction algorithms as a building block.

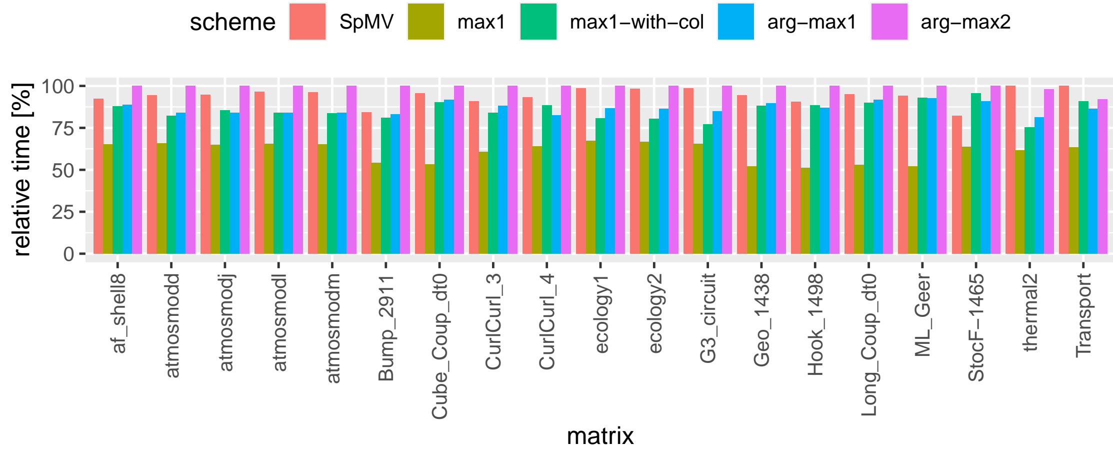


Figure 26.: SRCSR-adaptive with different type and operator instantiations for four other use cases in comparison to standard SpMV $y = \beta y + \alpha Ax$. The single precision matrices are listed in Table 11.

Our implementation does not contain any special branching for particularly long or short matrix rows and stands out because of it support for general types and operators while the performance of standard sparse matrix-vector operations competes with cuSPARSE. The generality enables a high flexibility in the usage of the kernels, which we used to find the maximum pairs or values and their column indices in each matrix row. Essential for the performance is the type configuration for data in different memory locations like global memory, shared memory, or thread local registers. Moreover, we showed another application for data transposition in shared memory to formulate a warp transposed segmented reduction algorithm, which runs as fast as implementations with CUB [84] within our sparse matrix-vector product benchmark.

3.8. Other Contributors

Falk Mayer refactored the source code and generalized the work adaptive scheme for arbitrary long matrix rows. Julius Ernesti and Holger Wünsche implemented the balanced-postprocess scheme. Bálint Soproni and Joachim Meyer implemented the warp transposed segmented reduction of Section 3.5.2.

4. Highly Parallel Linear Forest Extraction from a Weighted Graph on GPUs

Reference: This chapter was already published in [73].

Abstract For graph matching, each vertex is allowed to match with exactly one other vertex, such that the spanning subgraph of the matching has a maximum degree of one, i.e., the subgraph is a [0,1]-factor. In this work, we provide a highly parallel algorithm to extract a spanning subgraph with a maximum degree of n (the subgraph is a $[0, n]$-factor) and demonstrate the efficiency of our GPU implementation for $n = 1, 2, 3, 4$ by expressing the algorithm in terms of generalized sparse matrix-vector products. Moreover, from the [0,2]-factor, we compute a maximum linear forest (union of disjoint paths) by breaking up cycles and permuting the subgraph with respect to the vertex order within the paths. Both tasks execute efficiently on the GPU because of our novel parallel scan implementation, which does not require a random access iterator. As an application of linear forests, we demonstrate the algebraic creation of enhanced tridiagonal preconditioners for various large matrices from the Sparse Matrix Collection and report runtimes in the order of milliseconds for graphs with millions of edges and vertices on an RTX 2080 Ti.

4.1. Introduction

We recall some graph terminology [106]. An undirected graph of order N is a pair $G := (V, E)$ with a set of vertices $V \subset \mathbb{N}_0$ ($N := |V|$) and a set of edges $E := \{\{v, w\} \mid v, w \in V\}$. A *weighted* graph has in addition a function $\omega : V^2 \to \mathbb{R}$, which returns a weight $\omega(e) \neq 0$ for each edge $e \in E$ and zero otherwise. A *path* in G is a non-empty subgraph $P := (V_P, E_P) \subseteq G$ with distinct vertices v_i in $V_P = \{v_0, v_1, \dots, v_{k-1}\}$ and edges $E_P = \{\{v_0, v_1\}, \{v_1, v_2\}, \dots, \{v_{k-2}, v_{k-1}\}\}$. Vertices v_0 and v_{k-1} are called the *ends* of P. If the ends are connected with the additional edge $\{v_{k-1}, v_0\}$ then this is a *cycle* in G. A *spanning* subgraph of G is a graph with the same vertices as G but only a subset of its edges E.

In this chapter we are interested in the fast extraction of spanning subgraphs with certain properties. A $[0, n]$*-factor* [98] is a spanning subgraph of G in which each vertex has a degree of at most n. A $[0, 1]$-factor does not contain cycles, but all $[0, n]$-factors with $n \geq 2$ can. Removing cycles in general is hard. For the $[0, 2]$-factor we will present the fast removal of cycles to obtain an acyclic $[0, 2]$-factor. It is a collection of disjoint paths, also called a *linear forest*. Quickly reordering the vertices in the linear forest with respect to their order in the paths is a challenge, and we will discuss fast parallel algorithms for that.

Why are graph factors of interest? Classical graph matchings [24] compute $[0, 1]$-factors, which are used for optimizing the power consumption of wireless networks [120], preconditioning sparse linear systems [59], solving the data path allocation problem [21], and for the reordering and scaling of sparse matrices [64]. Computing maximum linear forests is the edge analog of the maximal path set problem [22], which is solved to

approximate the shortest superstring problem occuring during DNA sequencing [95]. Linear forests, which contain many strong edges, are also used for directional coarsening in algebraic multigrid [83], for adaptive algebraic smoothers [97], and for the setup of tridiagonal preconditioners, which we will discuss here as an application.

Tridiagonal systems can be inverted at the bandwidth limit of the GPU [72], so they lend themselves as fast preconditioners for a linear system of equations $Ax = d$. However, simply extracting the tridiagonal part of the system matrix A does not consider the strength of the included coefficients in the tridiagonal preconditioner and therefore gives suboptimal convergence rates. Instead, we want a tridiagonal preconditioner that contains many strong coefficients from A. We obtain it by extracting a linear forest from A and a permutation under which the adjacency matrix of the linear forest has a tridiagonal form.

This chapter consists of two major parts: first, formulation of the $[0, n]$-factor extraction (Section 4.3.2) in terms of generalized sparse matrix-vector products (Section 4.4.1), and second, the computation of a linear forest from a $[0, 2]$-factor by breaking up cycles and determining the path ID and position (Section 4.3.3). The two tasks in the second part run efficiently in parallel because of our novel parallel scan implementation, which does not require a random access iterator (Section 4.4.2). Section 5.3 presents benchmarks and Section 4.6 the application of enhanced tridiagonal preconditioners.

4.2. Related Work

$[0, 1]$-Factor Computations

Hagemann et al. [59] use weighted matchings to precondition symmetric indefinite linear systems, also similarly performed by Naim et al. [88] on GPUs. Cohen [24] and Naumov et al. [89] describe how graph matching and coloring is implemented efficiently on GPUs. The latter was also claimed in patents by Cohen et al. [15, 16] and Castonguay et al. [17]. Auer and Bisseling [44] developed a greedy graph matching on GPUs with an MD5 coloring technique to coarsen a graph in the context of graph partitioning. The graph matching problem is most related to our algorithms and was well studied on GPUs [24, 88, 89, 44], but only extracts matched vertex pairs instead of long paths from a graph.

Linear Forest Computations

Uehara and Chen [110] formulate three parallel algorithms for the calculation of maximal linear forests. The work of Shoudai and Miyano [105] shows that finding a maximal vertex-induced subgraph with a maximum degree of n is an NC^2 problem. These papers are theoretical, they do not provide actual parallel implementations.

GraphBLAS

the GraphBLAS standard [82] expresses graph problems in terms of linear algebra operations, e.g., a shortest-path calculation expressed as sparse matrix-vector multiplication on the semiring $\{\min, +, \mathbb{R} \cup \{+\}, +\}$. However, a memory efficient $[0, n]$-factor computation requires different types for the input and output vector, the sparse matrix, and the accumulator; this flexibility is supported by our generalized sparse matrix-vector multiplication.

Other Relations

Our graph algorithms are also related to linear sum assignment, and matrix transversal problems. The linear sum assignment problem [14, 13][1] is solved by a column permutation, which minimizes the sum of the diagonal entries of the column-permuted matrix, which generally represents a weighted bipartite graph, and was implemented on a GPU by Date and Nagi [25]. Maximum matrix transversals aim to provide a permutation, which maximizes the sum, product, or amount of non-zero entries of the diagonal elements of the permuted matrix, and was intensively studied by Duff et al. [37, 36, 38, 40, 39][2]. Both related problems can also be used to design algorithms, which extract one-dimensional subgraphs, but although they could provide long paths, they rely on parallel BFS algorithms [25] or consider dense matrices only [25, 58, 103].

4.3. Algorithms

We divide the algorithmic contributions of this chapter into two major parts: first, the computation of the $[0, n]$-factor π, and second, the extraction of a linear forest from a $[0, 2]$-factor.

4.3.1. Factor Notation

We generalize the notation from [44] and choose the following functional representation of the $[0, n]$-factor π:

$$\pi : V \to \Phi := \{\phi \in \mathcal{P}(V) \mid |\phi| \leq n\}, \quad \pi(v) \subseteq V_v, \tag{20}$$

$$\text{with } \; V_v := \{w \in V \mid \exists\, e \in E : \{v, w\} = e\ \} \setminus \{v\}, \tag{21}$$

and $\mathcal{P}(V)$ representing the power set of V. Thus, $\pi(v)$ returns either the empty set, one vertex, ..., or n vertices from the *neighborhood* V_v of v. Function π is subject to the following conditions:

1. For all $v \in V$ there exist at most n different vertices $w_i \in V \setminus \{v\}$ such that $v \in \pi(w_i)$ with $i = 0, \ldots, n-1$, i.e., we allow a vertex to have at most n neighboring vertices in the $[0, n]$-factor.

2. For all $v, w \in V$, $v \neq w$, if $v \in \pi(w)$, then $w \in \pi(v)$, $w \in V_v$, and $v \in V_w$, i.e., we only include existing edges $\{v, w\} \in E$ in the $[0, n]$-factor.

If it is not possible to increase the size of $\pi(V)$ further without breaking the conditions, the $[0, n]$-factor is *maximal*.

The *weight* of a $[0, n]$-factor ω_π is defined as

$$\omega_\pi := \sum_{e \in E_\pi} |\omega(e)|, \quad E_\pi := \{\{v, w\} \in E \mid v \in \pi(w)\}, \tag{22}$$

and the *relative weight coverage* c_π as

$$c_\pi := \frac{\omega_\pi}{\omega_G}, \quad \omega_G := \sum_{\{v,w\} \in E, v \neq w} |\omega(\{v, w\})|. \tag{23}$$

[1] An implementation is available in SciPy `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.linear_sum_assignment.html` or `https://bougleux.users.greyc.fr/lsape/`, which is described in [13].

[2] An implementation is available on `http://www.hsl.rl.ac.uk/catalogue/hsl_mc64.html`

Algorithm 6: Sequential greedy $[0, n]$-factor computation on a weighted graph $G = (V, E)$.

```
1 for v ∈ V do
2 |   π(v) ← ∅
3 end
4 for (v, w) ∈ E in order of decreasing |ω({v, w})| do
5 |   if |π(v)| < n, |π(w)| < n, and v ≠ w then
6 |   |   π(v) ← π(v) ∪ {w}
7 |   |   π(w) ← π(w) ∪ {v}
8 |   end
9 end
```

For comparisons with the original vertex ordering we also define

$$c_{\text{id}} := \left(\sum_{i=0}^{N-1} |\omega(\{i, i-1\})| + |\omega(\{i, i+1\})| \right) / \omega_G \tag{24}$$

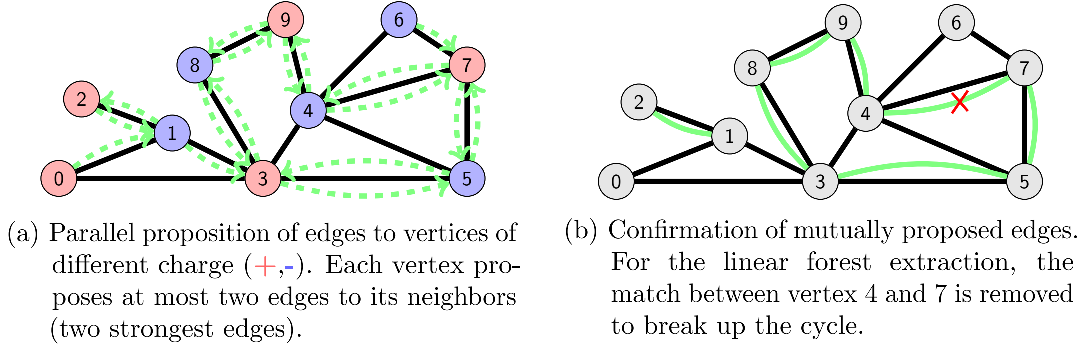
4.3.2. $[0, n]$-Factor Algorithms on Weighted Graphs

To evaluate the quality of our parallel $[0, n]$-factor algorithm, we use the sequential greedy $[0, n]$-factor Algorithm 6, which sorts the edges with respect to their weight in decreasing order and adds them to π if possible. Note that for $n = 1$, this algorithm computes a matching with at least half of the maximum weight considering all possible matchings [44].

Our parallel $[0, n]$-factor Algorithm 8 is structured similarly to the graph matching techniques of Auer and Bisseling [44], but contrary to classical graph matching, a vertex is allowed to have n other neighboring vertices in the $[0, n]$-factor. For each vertex at most n outgoing edges with the largest absolute weights are *proposed* in parallel in Algorithm 8 Line 19 and non-mutually proposed edges are removed in Line 27. The remaining edges are the *confirmed* edges and part of the $[0, n]$-factor. The neighboring set for potential propositions Θ in Line 19 excludes neighbors which already have n confirmed edges (Line 15), the same charge if $k \mod m \neq k_m$ (Line 17), and neighbors with existing confirmed edges (Line 19). If a vertex is charged prior to the edge proposition, it is positive(+) or negative(-) with a probability of p and $(1 - p)$, respectively, and is only allowed to propose to vertices with a different charge. The randomness of charge assignments enables larger $[0, n]$-factors for graphs with unfavorable structural edge weight properties, e.g., strict monotonically increasing edge weights in a specific direction. The charge of a vertex depends on its ID and the iteration index k.

The propositions and confirmations of edges are done iteratively with loop index k, to include more edges in the $[0, n]$-factor. The parameters m and k_m control when vertex charging is enabled, M represents an upper limit for the number of propositions, and the algorithm returns the number of actual propositions $M_{\max}$ for a maximal $[0, n]$-factor in Line 24.

Figure 27 shows the edge proposition and confirmation for charged vertices for $n = 2$. For each iteration step k, the charging of the vertices is disabled if $k \mod m = k_m$, such that each vertex is allowed to propose to every neighbor which did not reach the maximum number of confirmed edges yet. This has two advantages: first, for some graphs, the unrestricted edge proposition creates large $[0, n]$-factors after the first proposition step. If vertex charging had been used for these graphs, more iterations would have been required



(a) Parallel proposition of edges to vertices of different charge (+,-). Each vertex proposes at most two edges to its neighbors (two strongest edges).

(b) Confirmation of mutually proposed edges. For the linear forest extraction, the match between vertex 4 and 7 is removed to break up the cycle.

Figure 27.: Edge proposition and confirmation for a $[0, 2]$-factor (Algorithm 8, with $n = 2$, $k = 0$, $k_m = 0$) executed on a small graph.

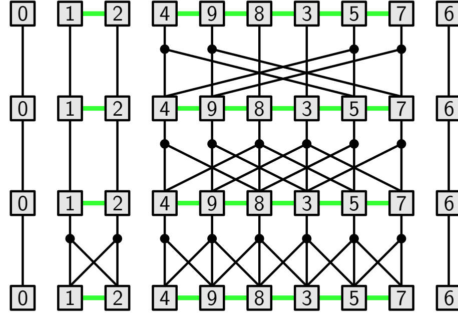


Figure 28.: Three steps (bottom to top) of a parallel bidirectional scan for the graph of Figure 27 with $N = 10$ vertices and 4 paths. Vertices of the same path are connected with green horizontal lines. Each step represents one kernel launch.

to obtain the same $[0, n]$-factor size. Second, if nothing is proposed without vertex charging enabled, the $[0, n]$-factor is maximal and we may stop the iterations (Alg. 8 Line 23).

4.3.3. From a $[0, 2]$-Factor to a Linear Forest

The $[0, 2]$-factor represented by π contains only connectivity information. Only the two or fewer neighbors are known for each vertex, whereas it is unknown in which path or cycle a vertex is located (there is no path ID), nor at which position within the path or cycle a vertex resides. We first break up the cycles. For that purpose, the weakest edge of each cycle is removed to keep the weight ω_π (Eq. 22) of the linear forest large. Afterward, we compute the path ID and position within the path for each vertex. A permutation of the adjacency matrix of the linear forest, which makes it tridiagonal, is obtained by sorting the vertex IDs with respect to their key composed of path ID and position.

We arrange the linear forest extraction algorithm from a $[0, 2]$-factor π into four *steps*:

1. **Identify cycles:** identify cycles and break them up by removing their weakest edge.

2. **Identify paths:** obtain the path ID and position within the path for all vertices.

3. **Compute permutation:** sort vertex IDs with respect to their path ID and position to get the permutation in which the adjacency matrix of the linear forest is tridiagonal.

4. **Extract weight coefficients:** with the permuattion, extract coefficients from the adjacency matrix of G.

Algorithm 7: Parallel bidirectional scan to compute path IDs and positions for a linear forest.

```
   Data: acyclic [0,2]-factor π
   Result: path IDs l, and positions p
 1 for v ∈ V do in parallel // kernel launch
 2 |   r(v) ← (1,1)
 3 |   q(v) ← make_tuple(π(v))
 4 end
 5 while ∃ w_i ∈ q(v) for any v ∈ V such that not is_path_end(w_i) do
 6 |   q' ← q // copy
 7 |   r' ← r // copy
 8 |   for v ∈ V do in parallel // kernel launch
 9 |   |   (w_0, w_1) ← q'(v)
10 |   |   (r_0, r_1) ← r'(v)
11 |   |   for i = 0,1 do
12 |   |   |   if not is_path_end(w_i) then
13 |   |   |   |   (v_0, v_1) ← q'(w_i)
14 |   |   |   |   (t_0, t_1) ← r'(w_i)
15 |   |   |   |   for j = 0,1 do
16 |   |   |   |   |   if v_j ≠ v then
17 |   |   |   |   |   |   r_i ← r_i + t_j
18 |   |   |   |   |   |   w_i ← v_j
19 |   |   |   |   |   end
20 |   |   |   |   end
21 |   |   |   end
22 |   |   end
   |   |   // write updated values
23 |   |   q(v) ← (w_0, w_1)
24 |   |   r(v) ← (r_0, r_1)
25 |   end
26 end
   // chose one path end as path ID
27 for v ∈ V do in parallel // kernel launch
28 |   (w_0, w_1) ← q(v)
29 |   (r_0, r_1) ← r(v)
30 |   i ← arg min_{j∈{0,1}}(w_j)
31 |   l(v) ← w_i
32 |   p(v) ← r_i
33 end
```

Algorithm 8: Parallel $[0, n]$-factor algorithm. Function `charge` returns positve(+) or negative(-).

```
1  Function max_target(v, Θ)
2  |  return arg max_{w∈Θ} (|ω({v, w})|)
3  end
   Data: Weighted graph G = (V, E), ω, and integers m, k_m
   Result: Number of iterations M_max and [0, n]-factor π
4  for v ∈ V do in parallel
5  |  π(v) ← ∅
6  end
7  for k = 0, ..., M − 1 do
8  |  if k mod m ≠ k_m then
9  |  |  for v ∈ V do in parallel // kernel launch
10 |  |  |  q(v) ← charge(v,k)
11 |  |  end
12 |  end
13 |  π′ ← π // copy current π
14 |  for v ∈ V do in parallel // kernel launch
15 |  |  W ← V_v \ {w ∈ V | |π′(w)| = n}
16 |  |  if k mod m ≠ k_m then
17 |  |  |  W ← W \ {w ∈ V | q(w) = q(v)}
18 |  |  end
   |  |  // Propose edges to neighbors
19 |  |  while |π(v)| < n and |Θ ← W \ π(v)| > 0 do
20 |  |  |  π(v) ← π(v) ∪ {max_target(v, Θ)}
21 |  |  end
22 |  end
   |  // [0, n]-factor is maximal?
23 |  if |π(V)| = |π′(V)| and k mod m = k_m then
24 |  |  return k + 1 // return M_max
25 |  end
   |  // Remove non-mutual propositions
26 |  for v ∈ V do in parallel // kernel launch
27 |  |  π(v) ← {w ∈ π(v) | v ∈ π(w)}
28 |  end
29 end
30 return M
```

For steps (1) and (2), we use a bidirectional scan to design parallel algorithms. A *bidirectional* scan executes two scans in two opposite directions simultaneously. In this way, we can compute the result of two different opposed scans or find and broadcast a specific value in parallel. The access pattern of our bidirectional scan on multiple paths is shown in Figure 28. Considering a single path, this pattern appears in the Parallel Cyclic Reduction Algorithm [31]. Such an access pattern allows, for example, to broadcast a value to all threads participating in the scan, although a single thread does not visit every neighbor explicitly. The scan must be bidirectional because π is structured like a double-linked list but with unknown orientation about which neighbor is forward and which backward, e.g., within the $[0, 2]$-factor, the forward neighbor of vertex 8 in the right part of Figure 27 might be vertex 9, but vertex 4 might be the backward neighbor of vertex 9.

In the bidirectional scan for step (2), the path-ends and the path-positions for all vertices are determined, which is shown in Algorithm 7. For the latter, each vertex initializes its forward and backward oriented position with a '`1`' in Line 2 and the bidirectional scan with an addition operator, which is applied in Line 17, computes the path position in both directions. The initialization of the stride-q neighbors in Line 3, sets $q(v)$ to the $[0, 2]$-factor neighbors $\pi(v)$, and fills up the tuple with the vertex ID v, but marked as a path end. First, q saves the vertices, which are visited in the next scan iteration step but also contains both path ends after execution, which is assigned to the result in Line 31. We define the path ID as the minimum ID of the vertices at the path ends, and this defines also the orientation: the vertex at the path end with the smaller ID is at position 1, its neighbor at position 2, etc.

Scan algorithms are often parameterized on the operation (e.g., `thrust::inclusive_scan` [62]) so that prefix sums and other properties like minima can be computed by changing the operator to `min`. In the same fashion, we use a bidirectional scan for step (1) to determine the weakest edge within a cycle. The weakest edge is uniquely identified by the weight and the IDs of the incident vertices. The algorithm for step (1) is constructed analogously to step (2) but only identifies cycles and their weakest edge.

Theoretically, steps (1) and (2) can be merged by searching for the weakest edge and the distance to it, but in practice this incurs more data movement and longer running times.

4.4. Implementation

All implementations use CUDA for a parallel execution on NVIDIA GPUs. In the following, we assumed that the graph is saved as an adjacency matrix A. Thus, each matrix entry a_{ij} corresponds to the weight of the edge between vertex i and vertex j. To avoid additional branching in the kernels the diagonal of A is deducted and the coefficients are set to their absolute values with $A' := |A| - \texttt{diag}(|A|)$ before the $[0, n]$-factor computation. The implementation of Algorithm 8 also supports directed input graphs for the calculation of π, which works well for the test cases presented in this chapter. However, constructing π from an underlying undirected graph and extracting the coefficients from the original graph is a better alternative for general graphs.

4.4.1. Parallel $[0, n]$-factor computation for $n \leq 4$

The edge proposition (Alg. 8, Line 14) was implemented by leveraging a *generalized* sparse matrix-vector on the GPU, where the multiplication is replaced by a different binary operation ($\otimes$) and the summation is replaced by a different reduction operation ($\oplus$). For the first edge proposition ($k = 0$), a reduction-by-key algorithm is sufficient to find the n maximum edge weights in each row i of matrix A', which is shown in Table 13 for vertex

	read			written		
	label	length	type	label	length	type
for $k = 0$	CSR values	`nnz`	value	proposed edges	nN	index
	CSR col indices	`nnz`	index	proposed edge weights	nN	value
	CSR row ptrs	$N + 1$	index			
	vertex charges	N	bool			
additional data for $k > 0$	confirmed edges	nN	index			

Table 12.: Read and written buffers in global GPU memory for the implementation of the edge proposition, which is expressed as generalized sparse matrix-vector product.

4, $n = 2$ and A' in CSR format. The corresponding accumulator type saves n sorted pairs of a value and its corresponding column index j. In the beginning of the example the first value-column pair $(0.2, 3)$ is the initial value of the accumulator, and pairs with larger coefficients are inserted into the accumulator in subsequent steps to the right. With vertex charging enabled, the coefficients of the same charge as the current matrix row are ignored, which results in a proposition of vertex 4 to vertices 9 and 7 because these edges are of maximum weight and different charge. Without charging, vertex 4 proposes to vertices 6 and 9.

In subsequent edge propositions ($k > 0$), a vertex is allowed to propose edges to vertices, which do not have n confirmed edges yet. That indirect lookup is expressed in sparse matrix-vector products $A'x$ by vector x, which saves the $n \cdot N$ confirmed edges. If a vertex j has already n confirmed edges, the result of the abstract multiplication operator $(A')_{i,j} \otimes x_j$ is equal to zero, and the edge is ignored during edge proposition. When vertex charging is enabled, another additional indirect lookup in the abstract multiplication operator ensures that zero is returned if the vertices have the same charge. The charges are calculated before each edge proposition in Line 10 with a part of the MD5 algorithm, which was also used by Auer and Bisseling [44]. We summarize the memory requirements for the edge proposition in Table 12. Note that the edge weights of A' are also written if $n = 2$, such that the minimum edge weight of cycles can be identified in the subsequent Algorithm 7.

After the edge proposition, the parallel loop in Algorithm 8 Line 26 is executed by another kernel to remove non-mutually proposed edges.

4.4.2. From a $[0, 2]$-Factor to a Linear Forest

We implemented a step-efficient bidirectional scan to obtain the path IDs and position of the vertices with $\log_2(N)$ kernel launches and a butterfly access pattern, which is visualized in Figure 28 for a linear forest. Instead of explicitly checking the condition in Algorithm 7 Line 5, the kernel is launched $\log_2(N)$ times, so even if all vertices reside in one path we obtain the correct result.

If we reach the path's end, during the stride-q neighbor computation, we mark this by setting the stride-q neighbor to the negative 1-based index of the path's end ID. Let $q_{\max}$ be the last and largest stride of the algorithm. A positive stride-$q_{\max}$ neighbor after $\log_2(N)$ scan steps indicates that the vertex is part of a cycle because had it reached a path's end, it would be negative.

The first bidirectional scan of step (1) in Section 4.3.3 requires read/write buffers for the stride-q neighbors, the weakest edge weights, and the vertex IDs incident to the weakest edge. The second bidirectional scan for step (2) in Section 4.3.3 requires only read/write

$((A')_{4,j},\ j)$	(0.2, 3)	(0.3, 5)	(0.9, 6)	(0.4, 7)	(0.5, 9)
accumulator without charging	$(0.2, 3)$ $(0.0, _)$	$(0.3, 5)$ $(0.2, 3)$	$(0.9, 6)$ $(0.3, 5)$	$(0.9, 6)$ $(0.4, 7)$	$(0.9, 6)$ $(0.5, 9)$
charge	+	-	-	+	+
accumulator with charging	$(0.2, 3)$ $(0.0, _)$	$(0.2, 3)$ $(0.0, _)$	$(0.2, 3)$ $(0.0, _)$	$(0.4, 7)$ $(0.2, 3)$	$(0.5, 9)$ $(0.4, 7)$

Table 13.: Edge proposition for vertex 4 (-) of Figure 27 expressed as reduction along matrix row $(A')_{4,j}$ from left to right. The accumulator consists of two ($n = 2$) sorted pairs $((A')_{i,j},\ j)$.

buffers for the stride-q neighbors and the path positions, thus requiring less memory.

Each buffer mentioned above is allocated twice as an input and output buffer and used in a ping-pong fashion. Otherwise, other threads might read a value of a neighboring vertex during the scan execution while the updated result for that vertex has already been written to memory.

The above algorithms could not have been implemented with other scan operators of GPU libraries like Thrust [62] or CUB [86] as these are restricted to random access iterators. Even on CPUs, all parallel implementations of reduction or scan always assume random access iterators, cf. parallel STL in C++17. Our novel parallel scan implementation does not have this restriction; bidirectional connectivity suffices. This is even weaker than the concept of a bidirectional iterator, which includes global orientation information of what is forward and backward. We only have bidirectional connectivity, not knowing which neighbor is forward and which is backward along the path and still compute the scan in parallel.

4.4.3. Permute and Extract a Linear Forest

To obtain a permutation Q of A such that the edge weights, which are part of a linear forest of A are located in the tridiagonal part of $Q^T AQ$, the vertex IDs are sorted with a radix sort from CUB [84] with respect to their key composed of path ID and position. The coefficients of the tridiagonal system are taken from the original input matrix A by converting A into a COO format and assigning one GPU thread to one coefficient of A. With the vector of the confirmed edges, each thread checks if the edge is part of the linear forest and scatters its value with the permutation into the tridiagonal system, which is saved in three buffers of length N.

4.5. Results

For the results presented in this chapter, we use a machine with CentOS 7, CUDA Toolkit 11.4.48, CUDA driver 470.74, GCC 10.2.0, a GeForce 2080 Ti and an Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz. If not mentioned explicitly, the experiments were done in single precision as the RTX 2080 Ti only has a few double precision units. Additionally to the test matrices from the Sparse Matrix Collection [27], which are listed in Table 14 we use three 2D anisotropic problems (ANISO1,2,3) from [72], which represent an equidistant grid with the stencils:

matrix	symmetric	N	nnz	$\overline{\Delta(G)}$
AF_SHELL8	y	504 855	17 588 875	34.84
ANISO1	y	6 250 000	56 220 004	9.00
ANISO2	y	6 250 000	56 220 004	9.00
ANISO3	y	6 250 000	56 220 004	9.00
ATMOSMODD	n	1 270 432	8 814 880	6.94
ATMOSMODJ	n	1 270 432	8 814 880	6.94
ATMOSMODL	n	1 489 752	10 319 760	6.93
ATMOSMODM	n	1 489 752	10 319 760	6.93
BUMP_2911	y	2 911 419	127 729 899	43.87
CUBE_COUP_DT0	y	2 164 760	127 206 144	58.76
CURLCURL_3	y	1 219 574	13 544 618	11.11
CURLCURL_4	y	2 380 515	26 515 867	11.14
ECOLOGY1	y	1 000 000	4 996 000	5.00
ECOLOGY2	y	999 999	4 995 991	5.00
G3_CIRCUIT	y	1 585 478	7 660 826	4.83
GEO_1438	y	1 437 960	63 156 690	43.92
HOOK_1498	y	1 498 023	60 917 445	40.67
LONG_COUP_DT0	y	1 470 152	87 088 992	59.24
ML_GEER	n	1 504 002	110 879 972	73.72
STOCF-1465	y	1 465 137	21 005 389	14.34
THERMAL2	y	1 228 045	8 580 313	6.99
TRANSPORT	n	1 602 111	23 500 731	14.67

Table 14.: Pattern symmetric test matrices which are either from the Sparse Matrix Collection [27] or taken from [72] (ANISO{1,2,3}). $\overline{\Delta(G)}$ denotes the mean degree of the graph G.

$$\overset{\text{ANISO1}}{\begin{pmatrix} -0.2 & -0.1 & -0.2 \\ -1.0 & 3.0 & -1.0 \\ -0.2 & -0.1 & -0.2 \end{pmatrix}}, \quad \overset{\text{ANISO2}}{\begin{pmatrix} -0.1 & -0.2 & -1.0 \\ -0.2 & 3.0 & -0.2 \\ -1.0 & -0.2 & -0.1 \end{pmatrix}},$$

and matrix ANISO3 is obtained by permuting ANISO2, such that the coefficients with value -1.0 are located on the sub- and superdiagonal of A.

4.5.1. Weight Coverage Results

Table 15 shows the weight coverage results for the parallel $[0,2]$-factor computation with Algorithm 8 in comparison to the greedy sequential Algorithm 6. When A' is not symmetric, the $[0,n]$-factor computations use $A' + A'^T$, but the weight coverage results are calculated with respect to the original matrix A. For the parallel $[0,2]$-factor computation, we use three different *configurations*:

1. $m = 1,\ k_m = 0$: no vertex charging enabled $\forall k$.
2. $m = 5,\ k_m = 0$: no charging on iterations $k = 0, 5, 10, \ldots$.
3. $m = 5,\ k_m = 1$: no charging on iterations $k = 1, 6, 11, \ldots$.

All configurations use $p = 0.5$, which is the rounded optimal value for graph matching determined by Auer and Bisseling [44]. For configuration (1), the weight coverage of the maximal $[0,2]$-factor exceeds the results of the sequential algorithm for matrices ECOLOGY1,2, ATMOSMODD, and ATMOSMODJ (Table 15). However, the number of iterations

to reach a maximal $[0, 2]$-factor is often high and the weight coverage increase per iteration is very little for the same matrices, which is indicated by low values of the weight coverage after five iterations $c_\pi(5)$. Configurations (2) and (3) perform much better on these matrices. The comparison between them points out the possible limitations of c_π if vertex charging is applied in the first iteration, e.g., for matrices STOCF-1465, G3_CIRCUIT, LONG_COUP_DT0 and the other matrices with red highlighted results. The same limiting effect is also observable for $n = 1, 3, 4$. Therefore, for all following results, we utilize configuration (2) with $M = 5$ as the default configuration because it results in the same weight coverage as the sequential $[0, 2]$-factor algorithm in most cases.

	parallel alg. 8									**sequential alg. 6**
	no charging $\forall k$			no charging on $k = 0, 5, \ldots$			no charging on $k = 1, 6, \ldots$			
matrix	$c_\pi(5)$	$c_\pi(M_{\max})$	$M_{\max}$	$c_\pi(5)$	$c_\pi(M_{\max})$	$M_{\max}$	$c_\pi(5)$	$c_\pi(M_{\max})$	$M_{\max}$	c_π
AF_SHELL8	0.20	0.24	195	0.23	0.23	16	0.22	0.22	17	0.23
ANISO1	0.67	0.67	1252	0.67	0.67	11	0.54	0.54	17	0.67
ANISO2	0.67	0.67	1251	0.67	0.67	11	0.57	0.57	12	0.67
ANISO3	0.67	0.67	55	0.67	0.67	11	0.56	0.56	17	0.67
ATMOSMODD	0.02	0.47	164	0.41	0.42	16	0.42	0.42	17	0.44
ATMOSMODJ	0.02	0.47	164	0.41	0.42	16	0.42	0.42	17	0.44
ATMOSMODL	0.48	0.49	297	0.49	0.49	16	0.43	0.43	12	0.49
ATMOSMODM	0.95	0.95	297	0.95	0.95	16	0.74	0.74	12	0.95
BUMP_2911	0.81	0.82	31	0.81	0.82	26	0.64	0.64	27	0.82
CUBE_COUP_DT0	0.26	0.26	102	0.26	0.26	21	0.22	0.22	22	0.26
CURLCURL_3	0.34	0.34	47	0.34	0.34	16	0.36	0.36	12	0.34
CURLCURL_4	0.33	0.34	47	0.33	0.33	16	0.35	0.35	12	0.34
ECOLOGY1	0.00	0.50	1037	0.46	0.47	16	0.46	0.47	17	0.47
ECOLOGY2	0.00	0.50	1038	0.46	0.47	16	0.46	0.47	17	0.47
G3_CIRCUIT	0.56	0.71	159	0.70	0.70	16	0.59	0.59	17	0.70
GEO_1438	0.28	0.28	18	0.28	0.28	16	0.25	0.25	17	0.28
HOOK_1498	0.22	0.22	11	0.22	0.22	16	0.20	0.20	17	0.22
LONG_COUP_DT0	0.70	0.70	110	0.69	0.69	31	0.55	0.55	27	0.70
ML_GEER	0.20	0.20	383	0.20	0.20	11	0.17	0.17	17	0.20
STOCF-1465	1.00	1.00	11	1.00	1.00	16	0.78	0.78	17	1.00
THERMAL2	0.47	0.47	7	0.47	0.47	16	0.44	0.44	12	0.47
TRANSPORT	0.24	0.49	290	0.45	0.45	16	0.44	0.44	17	0.47

Table 15.: $[0, 2]$-factor computation on the undirected graph of the given matrix and their relative weight coverage (Eq. 23) after five iterations $M = 5$, $c_\pi(5)$, and the maximal $[0, 2]$-factor $c_\pi(M_{\max})$, which is reached after $M_{\max}$ iterations.

In Table 16 the weight coverages for $n = 1, 2, 3, 4$ are shown with the previously chosen default parameters in comparison to the sequential results. With the maximum difference between the parallel and sequential result of 0.04 for $n = 1$ and matrix ATMOSMODM, the parallel algorithm reaches almost the same weight coverage as the sequential algorithm. Additionally, the table contains the weight coverage of the sub- and superdiagonal c_{id} given by Equation 24. Comparing c_{id} with $c_\pi(5)$ for $n = 2$ allows an estimation of the weight of an algebraically extracted tridiagonal system with the tridiagonal part of A in the original vertex order, e.g., the tridiagonal part of ATMOSMODD already contains strong coefficients, whereas the algebraically extracted $[0, 2]$-factor of ATMOSMODM holds a much larger weight than just the tridiagonal part of the matrix in the original order.

4.5.2. Performance Results

Edge Proposition of Parallel $[0, n]$-Factor Computation

We use our generalized sparse matrix-vector implementation for the parallel edge proposition of Algorithm 8, Line 14 and compare the performance with the normal cuSPARSE SpMV and our segmented reduction (SRCSR) SpMV implementation, which both calculate $d = Ax + d$ for a CSR matrix. The implementation of the parallel edge proposition and the SRCSR SpMV schemes only differ by data types and functors, and use the same generic sparse matrix-vector API. The time of the SpMV setup kernels for cuSPARSE and our generalized sparse matrix-vector implementation is not included in the time measurement. Although the setup of cuSPARSE's SpMV is not explicitly exposed, a binary search kernel

matrix	c_{id}	$c_\pi(5)$ $n=1$ PAR	$n=1$ SEQ	$n=2$ PAR	$n=2$ SEQ	$n=3$ PAR	$n=3$ SEQ	$n=4$ PAR	$n=4$ SEQ	2x2 block tridiagonal $m=1$	2x2 block tridiagonal $m=5$
AF_SHELL8	0.01	0.14	0.14	0.23	0.23	0.34	0.34	0.40	0.40	0.38	0.43
ANISO1	0.68	0.27	0.29	0.67	0.67	0.72	0.73	0.79	0.79	0.68	0.64
ANISO2	0.13	0.27	0.29	0.67	0.67	0.72	0.73	0.79	0.79	0.68	0.64
ANISO3	0.68	0.27	0.29	0.67	0.67	0.72	0.73	0.79	0.79	0.68	0.64
ATMOSMODD	0.46	0.19	0.21	0.41	0.44	0.65	0.67	0.93	0.93	0.02	0.50
ATMOSMODJ	0.46	0.19	0.21	0.41	0.44	0.65	0.67	0.93	0.93	0.02	0.50
ATMOSMODL	0.25	0.21	0.22	0.49	0.49	0.60	0.61	0.73	0.73	0.41	0.45
ATMOSMODM	0.03	0.38	0.42	0.95	0.95	0.96	0.96	0.97	0.97	0.94	0.86
BUMP_2911	0.01	0.46	0.49	0.81	0.82	0.84	0.84	0.86	0.86	0.84	0.83
CUBE_COUP_DT0	0.06	0.11	0.13	0.26	0.26	0.33	0.34	0.38	0.38	0.29	0.29
CURLCURL_3	0.15	0.17	0.17	0.34	0.34	0.54	0.55	0.76	0.76	0.44	0.54
CURLCURL_4	0.15	0.17	0.17	0.33	0.34	0.53	0.54	0.74	0.74	0.40	0.53
ECOLOGY1	0.50	0.21	0.23	0.46	0.47	0.71	0.71	1.00	1.00	0.00	0.55
ECOLOGY2	0.50	0.21	0.23	0.46	0.47	0.71	0.71	1.00	1.00	0.00	0.55
G3_CIRCUIT	0.29	0.50	0.51	0.70	0.70	0.83	0.84	1.00	1.00	0.61	0.73
GEO_1438	0.04	0.13	0.14	0.28	0.28	0.36	0.37	0.44	0.44	0.33	0.33
HOOK_1498	0.04	0.11	0.11	0.22	0.22	0.28	0.28	0.33	0.33	0.25	0.25
LONG_COUP_DT0	0.10	0.49	0.50	0.69	0.70	0.79	0.79	0.87	0.87	0.84	0.83
ML_GEER	0.05	0.09	0.09	0.20	0.20	0.25	0.26	0.32	0.32	0.23	0.26
STOCF-1465	0.23	0.92	0.93	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
THERMAL2	0.10	0.23	0.24	0.47	0.47	0.68	0.68	0.84	0.84	0.58	0.58
TRANSPORT	0.49	0.20	0.22	0.45	0.47	0.68	0.70	0.98	0.98	0.25	0.53

Table 16.: $[0, n]$-factors of underlying undirected graphs and their relative weight coverages (Eq. 23) after five iterations $M = 5$, $c_\pi(5)$, $k_m = 0$, $m = 5$ of the parallel (PAR) Algorithm 8, in comparison to the results of the sequential (SEQ) Algorithm 6. The coverage of the sub- and superdiagonal in the original ordering is shown by c_{id} (Eq. 24). The weight coverage of the algebraically constructed 2x2 block tridiagonal preconditioner is shown in the right part of the table (see Section 4.6).

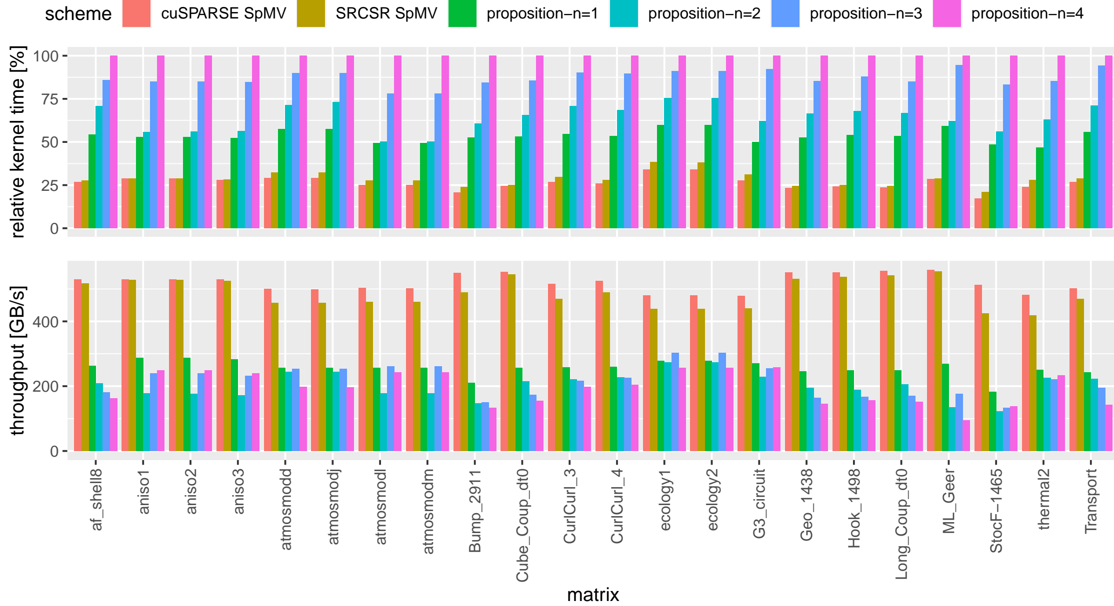


Figure 29.: Performance results for one kernel execution of edge proposition according to Algorithm 8, Lines 14-22 with $k > 0$, $m = 1$, $k_m = 0$ and different n in comparison to SpMV algorithms. Except cuSPARSE SpMV, all schemes use our generalized sparse matrix-vector implementation.

is executed prior to the actual sparse-matrix vector calculating kernel, which is visible with the profiler.

The average runtimes and throughputs were measured with NVIDIA Nsight Compute and are shown in Figure 29. Due to significant different matrix sizes, the upper plot shows the times relative to the longest kernel. For the edge proposition, amount of data which is read and written increases linearly with n (see Table 12) but the main limiting factor of the kernel is the reduction along the matrix rows. For a normal sparse matrix-vector product our general SRCSR code has similar performance to the specialized cuSPARSE assembly optimized code. Therefore, SRCSR has an efficient implementation, despite its generality which cuSPARSE does not have. The kernel (Algorithm 8, Line 14) is executed by SRCSR SpMV (Chapter 3) with appropriate lambda and type parameterization, but the lambdas contain a lot of complex code (70 lines). So the resulting SRCSR code after lambda inlining by the compiler

- does more work than a normal SpMV,
- has more instruction flow divergence (if-statements) than a normal SpMV,
- uses more registers than a normal SpMV,
- uses more shared memory than a normal SpMV,
- and uses more input and output vectors (more DRAM traffic) than a normal SpMV (see Table 12).

Therefore, we cannot expect Algorithm 8 to run at the speed of a normal SpMV. The performance of the normal SpMV serves as roofline in the comparison because it solves a simpler problem but on the same matrix structure. Achieving 30-50% of this roofline with

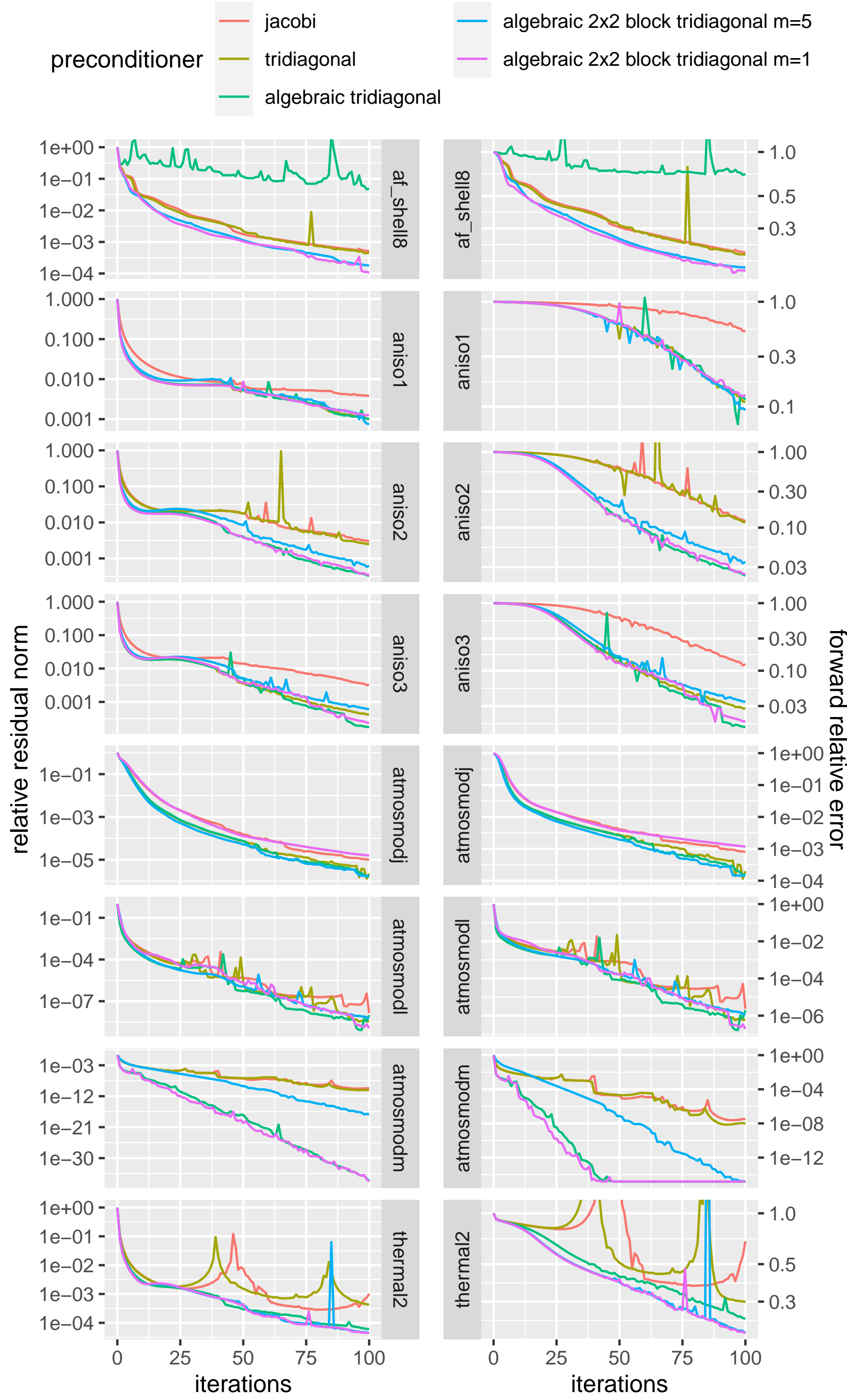


Figure 30.: Double-precision BiCGStab convergence results with our algebraically constructed scalar and 2x2 block tridiagonal preconditioner in comparison to a Jacobi and tridiagonal preconditioner based on the original vertex ordering.

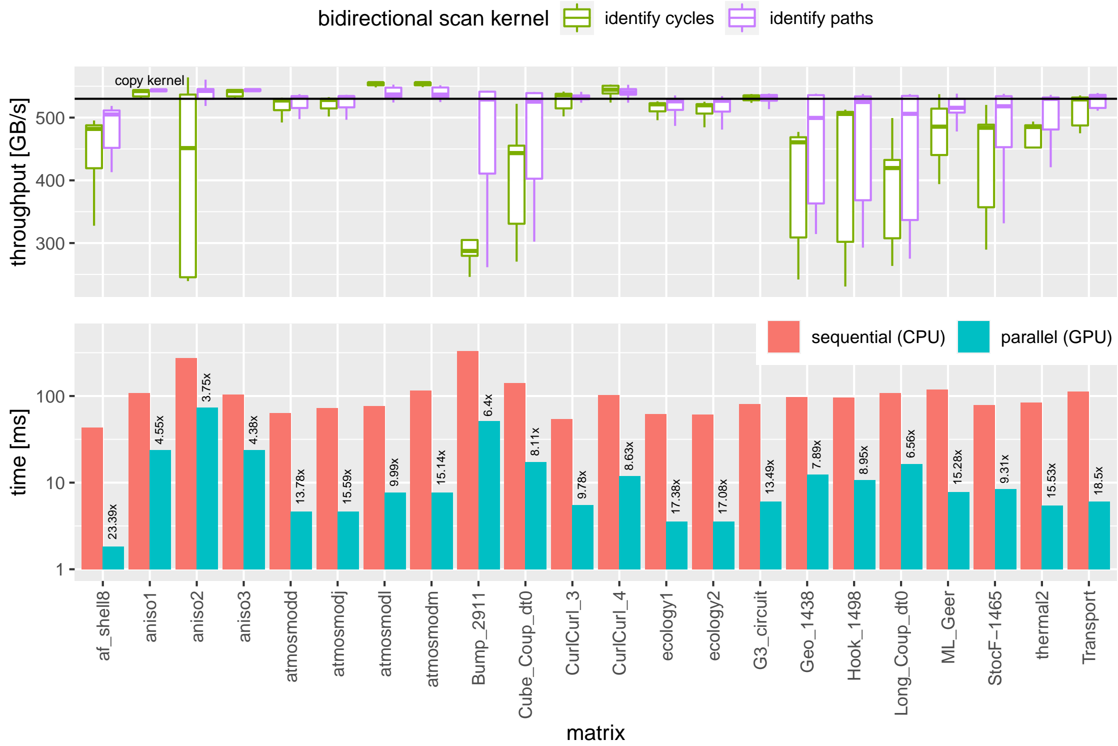


Figure 31.: Memory throughput statistics (top) for both bidirectional scan kernels and total runtimes (bottom) including steps (1), (2), (3) from Section 4.3.3. The speedup of the parallel version is written above the bars.

a more demanding and irregular algorithm (Algorithm 8), proves high efficiency of the implementation.

We evaluated alternative implementations for $[0, n]$-factor computation in which the SRCSR kernel contains less work and has a similar runtime to normal SpMV. But this requires more substeps and other additional kernels. Although the additional kernels run at full bandwidth the approach presented in the paper has the shortest overall runtime. Other approaches to find the columns of the n maximal values within each matrix row with CUB's [84] segmented reduction or segmented sort are approximately one order of magnitude slower for $2 \leq n \leq 4$.

Bidirectional Scan to Extract the Linear Forest

The linear forest extraction consists of the identification of the cycles, the path identification, and the extraction of the coefficients (see Section 4.3.3). The first two steps are implemented with our bidirectional scan that is executed $\log_2(N)$ times for each step. Figure 31 shows the throughput statistics of the bidirectional scans as a boxplot, which reveals worse throughput for some kernel executions due to the expected irregular global memory accesses when visiting the stride-q vertex neighbors. However, in most cases, the median of the throughputs is close to the performance of a simple copy kernel. The comparison of the averaged total running times of the sequential CPU version with the parallel GPU version in the lower part of the plot reports speedups from factor 4x to 24x. Contrary to the parallel solution, the sequential version performs far less work: it creates the permutation while the vertices are visited without an explicit sorting.

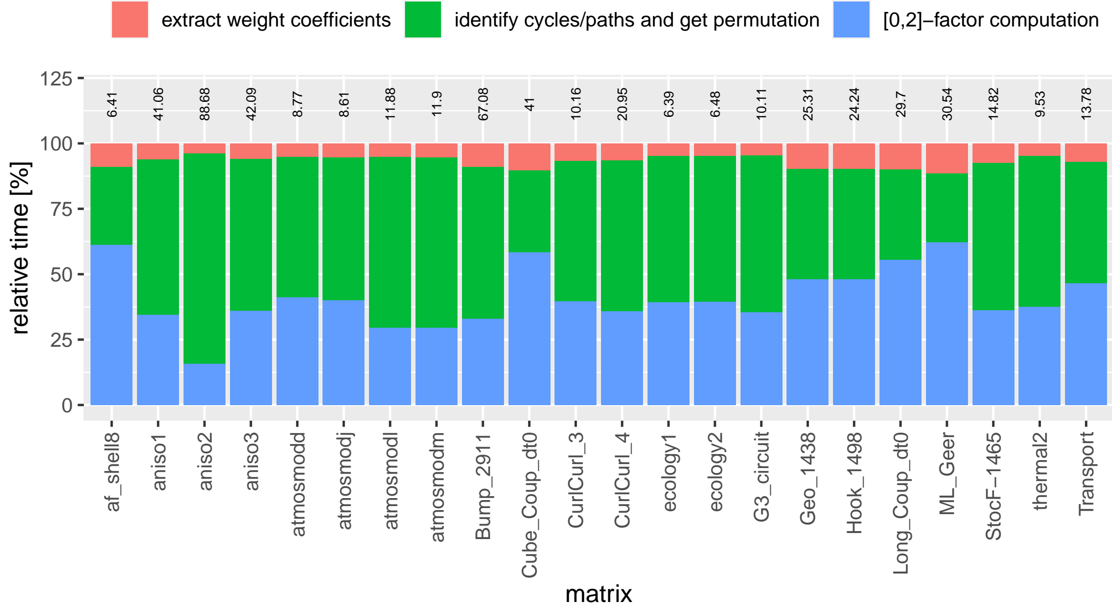


Figure 32.: Time breakdown for $[0,2]$-factor computation (Algorithm 8 with $M = 5, m = 5, k_m = 0, n = 2$), and the extraction of the linear forest (Section 4.3.3) to algebraically construct a tridiagonal preconditioner. The total absolute running time in milliseconds is written above the bars.

4.6. Application

As an exemplary application, we show the setup of an algebraic scalar (AlgTriScalPrecond) and 2x2 block tridiagonal preconditioner (AlgTriBlockPrecond). The $[0,2]$-factor computation and the extraction of a linear forest are used to setup AlgTriScalPrecond, which is shown in Figure 32 by the time breakdown of the complete setup. Algorithm 8 for the $[0,2]$-factor computation and the bidirectional scans (including Algorithm 7) consumes most of the time, whereas the actual coefficient extraction only requires at most 10% of the setup time.

AlgTriBlockPrecond is constructed by a $[0,1]$-factor and a subsequent $[0,2]$-factor computation. With the $[0,1]$-factor, the graph is coarsened, such that the matched pairs represent a single vertex in the coarser graph. On that coarse graph, the $[0,2]$-factor is computed, resulting in a 2x2 block tridiagonal system on the fine graph. For vertices without a match in the $[0,1]$-factor, we add an uncoupled ghost equation by setting the diagonal and right-hand side value in the corresponding additional row to one. Otherwise an irregular 2x2/1x1 block tridiagonal system would have to be processed. The weight coverage of AlgTriBlockPrecond is shown in the right part of Table 16 for $m = 1, 5$, which is used for both factor computations. For matrices ANISO1,2,3 and ATMOSMODM, $m = 1$ (no vertex charging $\forall k$) results into a higher weight coverage, whereas for matrices AF_SHELL8 and ECOLOGY1,2, $m = 5$ (no vertex charging on $k = 0, 5, 10, \dots$) is better. Hence we conclude that the parameters for a $[0,n]$-factor computation and for recursive $[0,n]$-factor computations on the coarser graphs, must be chosen differently to maximize the weight coverage but automatic parameter control in nested factor computations is beyond the scope of this thesis.

To evaluate the convergence of our new preconditioners, we compare it with a Jacobi and a tridiagonal preconditioner (TriScalPrecond) which are constructed based on the original

vertex ordering and use a BiCGStab as the outer Krylov solver [104]. The implementation of the outer solver and the Jacobi preconditioner is taken from the MAGMA [8] library, and for the solution of the (block) tridiagonal systems, we use the tridiagonal library, which is presented in this thesis in Chapter 2. The right-hand side is constructed from a generated solution with $x_t[i] := \sin(16\pi i/N)$. With the true solution being known, we calculate the forward relative error as FRE $:= |x - x_t|_2/|x_t|_2$, where x is the computed solution. The double-precision convergence results with the relative residual norm and the forward relative error are shown in Figure 30. The improvement of the algebraic tridiagonal preconditioners for matrix ANISO2 is expected, as they include the strong coefficients from the diagonal of the stencil, which were permuted manually to the sub- and superdiagonal in ANISO3. For matrices ATMOSMODJ, ATMOSMODL, and ATMOSMODM, the $[0, 2]$-factor contains increasingly more weight relative to the original vertex ordering c_{id}, which is visible in Table 16. The convergence improvement for matrix ATMOSMODM is strongest, as the algebraic preconditioners have a weight coverage of up to 95%, whereas TriScalPrecond has a weight coverage of only 3%. This example also shows the coupling between convergence rate and weight coverage of AlgTriBlockPrecond, which has either a weight coverage of 94% for $m = 1$ and performs as well as the AlgTriScalPrecond ($c_\pi(5) = 0.95$), or a weight coverage of 86% for $m = 5$ and performs worse. For matrix AF_SHELL8, the TriScalPrecond with a weight coverage of 1% includes approximately the same coefficients as the Jacobi preconditioner. The AlgTriScalPrecond with $c_\pi(5) = 0.23$ includes not enough off-diagonal coefficients to obtain a stable convergence behaviour, which is achieved by AlgTriBlockPrecond with a weight coverage of 38% or 43%. In summary, in most cases, we see significant benefits of algebraically creating the tridiagonal system (AlgTriScalPrecond) rather than relying on the tridiagonal part of the matrix in the original ordering. The block version (AlgTriBlockPrecond) performs consistently even better.

4.7. Conclusion

We have shown how to compute [0,n]-factors efficiently in parallel with our generalized sparse matrix-vector product and how our new bidirectional scan, which does not require a random access iterator, identifies cycles, paths and positions in a $[0, 2]$-factor to create a linear forest. Benchmarks on large graphs demonstrate our parallel algorithms' high weight coverage at high speed. In the application to linear equation systems the high coverage results in superior convergence of the algebraically constructed scalar and 2x2 block tridiagonal preconditioners.

5. Operator Split Preconditioners

Reference: Most parts of this chapter were already published in [74]. Figure 33 is new.

Abstract We present an algebraic framework for operator splitting preconditioners for general sparse matrices. The framework leads to four different approaches: two with alternating splittings and two with a multiplicative ansatz. The ansatz generalizes ADI and ILU methods to multiple factors and more general factor form. The factors may be computed directly from the matrix coefficients or adaptively by incomplete sparse inversions.

The special case of tridiagonal splittings is examined in more detail. We extract line segments from the adjacency graph of the sparse matrix and split the matrix according to these segmentations. We obtain specialized variants of the four general approaches. Parallel implementations for almost all steps are provided on a GPU. We demonstrate the effectiveness and efficiency of these preconditioners combined with GMRES on various matrices.

5.1. Operator Splittings

This section examines algebraic splittings of the matrix operator, no further information is required. However, application specific knowledge about the matrix may be useful in the choice of the splitting parameters.

5.1.1. Introduction

Given a sparse linear equation system with an $N \times N$ matrix

$$Ax = b \tag{25}$$

basic iterative methods are typically derived from an additive matrix splitting $A = M + (A - M)$ with an invertible M, leading to the iterative defect correction with $x^0 := 0, k = 0, \ldots, k_{\text{last}} - 1$

$$x^{k+1} := M^{-1}(b - (A - M)x^k) = x^k + M^{-1}(b - Ax^k) \tag{26}$$

$$= Gx^k + M^{-1}b, \quad G := I - M^{-1}A\,. \tag{27}$$

The iterations may alternate between different splittings $\overline{M}_0, \ldots, \overline{M}_{m-1}$ resulting in the repetition $(k = 0, \ldots, k_{\text{last}} - 1)$ of the alternating corrections

$$\begin{aligned} x^{mk+1} &:= x^{mk+0} + \overline{M}_0^{-1}(b - Ax^{mk+0}) \\ &\vdots \\ x^{mk+m} &:= x^{mk+m-1} + \overline{M}_{m-1}^{-1}(b - Ax^{mk+m-1}) \end{aligned} \tag{28}$$

$$\text{with } \textit{alternating splittings } \overline{M}_0, \ldots, \overline{M}_{m-1}\,, \tag{29}$$

With the appropriate preconditioner $\overline{M}_{\text{ALT-i}}$ the alternating corrections can be expressed as a single correction

$$\overline{M}_{\text{ALT-i}} := A(I - \overline{G}_{\text{ALT-i}})^{-1}, \quad \overline{G}_{\text{ALT-i}} := \prod_{l=m-1}^{0} (I - \overline{M}_l^{-1} A) \tag{30}$$

$$x^{k+1} := x^k + \overline{M}_{\text{ALT-i}}^{-1}(b - Ax^k)\,, \tag{31}$$

but Eq. 28 is clearly preferable for execution. However, even then the application of $\overline{M}_{\text{ALT-i}}$ has the disadvantage of requiring $m-1$ additional multiplications with A and in practice bounding the spectral radius of $\overline{G}_{\text{ALT-i}}$ requires bounding the spectral radii of all factors $(I - \overline{M}_l^{-1} A)$, although this is not a necessary condition. Therefore, in addition to the alternating splittings $\overline{M}_l$, we also research *multiplicative operator splittings (MOS)* with the ansatz

$$M_{\text{MOS}} := T \prod_{l=m-1}^{0} M_l\,, \tag{32}$$

with an invertible diagonal matrix T and some invertible matrices M_l. Both $\overline{M}_l(A)$ and $M_l(A)$ depend on A in their construction, however, the application of *each* $\overline{M}_l(A)$, $l > 0$ requires an additional multiplication with A (Eq. 28), whereas the application of *all* $M_l(A)$ does not require an additional multiplication with A, leading to faster iterations

$$x^{k+1} := x^k + M_{\text{MOS}}^{-1}(b - Ax^k)\,. \tag{33}$$

5.1.2. Outer Solver

Instead of the simple iterative defect correction with $\overline{M}_{\text{ALT-i}}$ (Eq. 31) or M_{MOS} (Eq. 33) faster convergence is achieved by using the operator splitting preconditioners inside a more powerful outer solver. We will employ all preconditioners with GMRES [104]

$$x^{k+1} := \texttt{GMRESstep}^k(A, M)\,, \tag{34}$$

whereby the intermediate x-outputs are optional, because GMRES maintains an inner state from which it can compute just the last $x^{k_{\text{last}}}$ at the very end. We have the same choice as in Eq. 28, we can repeatedly $(k = 0, \ldots, k_{\text{last}} - 1)$ apply the different splittings $\overline{M}_l$ in alternating iterations of FGMRES [104] (the flexible GMRES variant which supports varying preconditioners)

$$\begin{aligned} x^{km+1} &:= \texttt{FGMRESstep}^{km+0}(A, \overline{M}_0) \\ &\vdots \\ x^{km+m} &:= \texttt{FGMRESstep}^{km+m-1}(A, \overline{M}_{m-1})\,. \end{aligned} \tag{35}$$

The intermediate x-outputs are only shown for analogy with Eq. 28. Combining the different GMRES iterations (Eqs. 34 and 35) with the alternating and multiplicative preconditioners results in four different approaches. Section 5.3.1 shows them in an overview together with other schemes which we evaluate in the results section.

5.1.3. Outline and Contributions

We have already seen our algebraic approach to operator splittings in Sections 5.1.1 and 5.1.2. Discussing related work we explain how our MOS ansatz generalizes ADI and ILU methods to multiple factors and more general factor form (Sections 5.1.4 and 5.1.5). The following subsections examine choices for diagonal preconditioners (Section 5.1.6), discuss invariance under diagonal scaling (Section 5.1.7) and define tools for sparsity patterns (Section 5.1.8). Based on this preparatory work we present the direct and adaptive construction of *multiplicative operator splittings (MOS)* for general sparse matrices (Sections 5.1.9 and 5.1.10). The main contribution of Section 5.1 is the formal unification and generalization of ADI and ILU methods. In contrast to ILU, the matrix factors in our MOS preconditioners are not restricted to triangular form and we may have more than two factors. In contrast to ADI, the matrix factors in our MOS preconditioners are not restricted to tridiagonal form and they do not require an underlying tensor product grid. We have a clear superset of preconditioners with many new possibilities.

Section 5.2 examines the special case of tridiagonal splittings and discusses the parallel GPU algorithms that implement the functionality. For a general graph, the parallel construction of a linear forest and of the corresponding permutation is shown in Chapter 4. Repeating this construction on a modified auxiliary graph we obtain a decomposition of the original graph (matrix) into multiple linear forests (Section 5.2.1). From this decomposition we create the direct and adaptive tridiagonal multiplicative preconditioners (Sections 5.2.2 and 5.2.3), and describe the efficient transitions between permutations (Section 5.2.4) associated with different linear forests. The contribution in Section 5.2 is the demonstration of one practical realization of the new possibilities opened up by Section 5.1, namely construction of tridiagonal operator splitting preconditioners for general sparse matrices. This is probably the simplest non-trivial possibility that goes beyond ILU and ADI. The main challenge here is the algebraic decomposition of a general graph into multiple (almost) disjoint linear forests by an efficient iterative process.

Section 5.3 compares convergence and performance results of the four approaches with tridiagonal operator splitting preconditioners, different diagonal and other preconditioners. When using our new preconditioners we already see clear benefits, in particular for the more difficult problems, even though we evaluate the simplest of the new possibilities opened up by Section 5.1.

5.1.4. Related Work

The Alternating Direction Implicit (ADI) method by Peaceman and Rachford [91] considers elliptic and parabolic PDEs on a 2D tensor product grid, where the elliptic operator splits into two directions, e.g. continuous Laplace $\Delta = \partial^2/\partial \mathrm{Y}^2 + \partial^2/\partial \mathrm{X}^2$ and discrete Laplace $A = L_1 + L_0$ with tridiagonal L_1, L_0, which allow fast inversion. Then $Ax = b$ is solved by iterating $x^0 := 0, k = 0, \ldots, k_{\text{last}} - 1$

$$
\begin{aligned}
(L_1 + \rho_k I)x^{k+\frac{1}{2}} &:= -(L_0 - \rho_k I)x^k + b \\
(L_0 + \rho_k I)x^{k+1} &:= -(L_1 - \rho_k I)x^{k+\frac{1}{2}} + b\,,
\end{aligned}
$$

with alternating acceleration factors $\rho_k > 0$. This is the same as Eqs. 26 and 27 with

$$\begin{aligned} G &:= (L_0 + \rho_k I)^{-1}(L_1 - \rho_k I)(L_1 + \rho_k I)^{-1}(L_0 - \rho_k I) \\ &= (L_0 + \rho_k I)^{-1}(L_1 + \rho_k I)^{-1}(L_1 - \rho_k I)(L_0 - \rho_k I) \\ M &:= A(I - G)^{-1} = \frac{1}{2\rho_k}(L_1 + \rho_k I)(L_0 + \rho_k I) \\ &= (T + (L_1 - T/m))\, T^{-1}\, (T + (L_0 - T/m)) \end{aligned}$$

which is a special case ($m = 2, T = 2\rho_k I$) of our MOS ansatz in Eq. 37. For two factors ($m = 2$) Douglas and Pearcy [92, 93] give convergence results including the more general case where $\rho_k I$ is replaced by a diagonal matrix $\rho_k F$. However, with a general F we do not obtain this simple M anymore, because F and L_1 do not commute. Although the improvements from $\rho_k F$ have been known for a long time [34], this is seldom used in ADI practice.

ADI methods are primarily used on 2D and 3D tensor product grids, so there is little consideration of higher order splittings. The Gauss-Seidel generalization to multiple spatial variables [33] is a special case of Eq. 28. For two factors ($m = 2$), performance aspects with respect to memory and operations are treated in detail for ADI and related methods (LOD1/2, SS1/2) [108] as are combinations of ADI with multigrid [76]. Douglas and Dupont [35] and Wachspress [114] give a more thorough treatise of ADI methods.

While ADI methods typically refer to splittings along spatial dimensions, approximate matrix factorization (AMF) [111, 121, 55] and operator splitting [65, 52] refer to general splitting schemes to split dimensions, linear from non-linear, fast from slow, different physics, different objective functions, etc. AMF and operator splitting methods focus on time-dependent PDEs or ODEs (e.g. resulting from spatial semi-discretization of a time-dependent PDE) when matrix A has the form $A = I + \tau L = I + \tau(L_0 + ... + L_{m-1})$, with τ being a controllable time-step or integration parameter, which greatly facilitates the splitting. In our setting we do not have this freedom. Moreover, almost all work considers only $m \leq 3$.

The main motivation behind splitting methods is the *simpler* and *faster* implicit treatment of the individual components, the typical example being tridiagonal solves on a tensor product grid. However, operator splitting methods do not require tensor product grids and other spatial discretizations may be used, what simpler means is then application specific, e.g. less fill-in in direct sparse solvers [123].

Section 5.1 discusses operator splittings with general alternating splittings $\overline{M}_l$ and general matrix factors M_l. In Section 5.2 we examine the special case of tridiagonal splittings and take advantage of fast tridiagonal solvers on the GPU [67, 90, 26, 69, 53, 2, 19, 18, 72]. For this purpose we partition the adjacency graph of the sparse matrix into disjoint paths (i.e. a *linear forest*). This problem has been previously addressed by Mavriplis [83] and by Philip and Chartier [97], but not with a fast parallel construction in mind. We use our own parallel GPU implementation for this purpose [73]. However, we need more than one linear forest. We want to decompose the entire graph (at least all of its strong edges) into multiple (almost) disjoint linear forests. Finding out how many linear forests are needed to cover a graph is known as the *linear arboricity* problem in graph theory [3, 61]. However, these works are theoretical and do not consider the problem in the approximative sense stated above. So we do not know of prior work computing the decompositions examined in this thesis.

5.1.5. Generalization of ILU Factorizations

The invertible diagonal matrix T appears only on the left in our MOS ansatz (Eq. 32), but without loss of generality we can express the matrix factors M_l in a form which highlights the symmetry of the ansatz with respect to T

$$M_l := (I + T^{-1}M'_l) = T^{-1}(T + M'_l) \tag{36}$$

$$M_{\text{MOS}} := T \prod_{l=m-1}^{0} M_l = (T + M'_{m-1})T^{-1}(T + M'_{m-2})T^{-1} \cdots T^{-1}(T + M'_0)\,, \tag{37}$$

where the only restriction on the matrices M'_l is that the resulting M_l are invertible. Now setting $m = 2$, M'_1 strictly lower triangular and M'_0 strictly upper triangular recovers the general ILU ansatz as a special case of Eq. 37

$$M_{\text{MOS}} = (T + M'_1)T^{-1}(T + M'_0) = (I + M'_1T^{-1})(T + M'_0)\,. \tag{38}$$

Therefore, the MOS preconditioners discussed further in Section 5.1 generalize ILU preconditioners to multiple factors ($m > 2$) and more general form of factors (Eq. 36).

The classical ILU construction which computes the coefficients of T, M'_l iteratively in dependence on the previously computed coefficients does not easily generalize to $m > 2$ with arbitrary M'_l. However, we can generalize the fixed-point ILU construction [23] to $m > 2$ to compute the coefficients of T, M'_l in our MOS ansatz (Eq. 37). So far, we did not see much improvement over the analytic choices from Section 5.1.9 despite the much longer construction, but this might be worth exploring further in the future.

5.1.6. Diagonal Preconditioners

We compare standard and new diagonal preconditioners

$$(\texttt{diagp}_0(A))_{i,i} := (\texttt{diag}(A))_{i,i} = A_{i,i} \tag{39}$$

$$(\texttt{diagp}_1(A))_{i,i} := \texttt{unit}(A_{i,i})\,\texttt{max}\left(|A_{i,i}|, \sum_{j:j\neq i} |A_{i,j}|\right) \tag{40}$$

$$(\texttt{diagp}_2(A))_{i,i} := \texttt{unit}(A_{i,i}) \sum_{j} |A_{i,j}| \tag{41}$$

$$(\texttt{diag}_{l1}(A))_{i,i} := A_{i,i} + \sum_{j:j\neq i} |A_{i,j}| \tag{42}$$

$$\texttt{unit}(x) := \begin{cases} 1 & \texttt{if}\ x = 0 \\ x/|x| & \texttt{else} \end{cases}\,. \tag{43}$$

The definitions allow for complex coefficients. The known Jacobi $\texttt{diag}(A)$ and $l1$-Jacobi $\texttt{diag}_{l1}(A)$ [10] may run into the problem of being singular even though A is not, whereas the new $\texttt{diagp}_1(A)$ and $\texttt{diagp}_2(A)$ never have this problem. Moreover, even if $\texttt{diag}(A), \texttt{diag}_{l1}(A)$ are invertible, the absolute value of one entry could be very small in comparison to the other, so their maximum per row is a much more stable choice in case of general sparse matrices.

The original definition of $\texttt{diag}_{l1}(A)$ [10] has also the problem that $\texttt{diag}_{l1}(+A)$ and $\texttt{diag}_{l1}(-A)$ are completely different, so that sign changes can have dramatic effects. Therefore, we show the homogeneous $\texttt{diagp}_2(A)$ instead of $\texttt{diag}_{l1}(A)$ in the results section. In comparison, $\texttt{diagp}_2(A)$ is a slightly scaled version of $\texttt{diagp}_1(A)$:

$$I \leq \texttt{diagp}_2(A)\,\left(\texttt{diagp}_1(A)\right)^{-1} \leq 2I\,. \tag{44}$$

5.1.7. Invariance under Diagonal Scaling

A desirable property of a preconditioner $M(A)$ as a function of A is to be *homogeneous under diagonal scaling* with any invertible diagonal matrix F from left, because then the preconditioned linear equation system is *invariant under diagonal scaling*

$$\Big(FAx\Big) = \Big(Fb\Big)\,, \qquad M(FA) = F\,M(A) \tag{45}$$

$$M(FA)^{-1}\Big(FAx\Big) = M(A)^{-1}Ax = M(A)^{-1}b = M(FA)^{-1}\Big(Fb\Big)\,. \tag{46}$$

For such a preconditioner $M(A)$ we may chose some invertible diagonal F and construct $M(FA)$ without loss of generality.

We assume that the alternating splittings $\overline{M}_l(A)$ from Eq. 29 are chosen such that they are homogeneous under diagonal scaling, then the same follows for $\overline{M}_{\text{ALT-i}}(A)$ (Eq. 30). The other preconditioners $\texttt{diagp}_1(A), M_{\text{MOS-d}}(A), M_{\text{MOS-a}}(A)$ are homogeneous under diagonal scaling by construction. In more detail, in the general ansatz (Eq. 37) the $M_l(A)$ are invariant (Eqs. 60 and 69) and $T(A)$ is homogeneous (Eqs. 67 and 69) under diagonal scaling.

5.1.8. Sparsity Patterns

In the construction of preconditioners for sparse matrices the sparsity pattern, i.e. the index set of the non-zero coefficients in the matrix, plays an important role. In this subsection we define some tools to select sparsity patterns and measure which coefficients they contain. First let us define some general sparsity patterns

$$S^{\texttt{diag}} := \{(i,j) \mid i = j\}\,, \quad S^{\texttt{tri}} := \{(i,j) \mid |i-j| \le 1\} \tag{47}$$

$$S^{\texttt{low}} := \{(i,j) \mid i \ge j\}\,, \quad S^{\texttt{low-s}} := \{(i,j) \mid i > j\} \tag{48}$$

and follow up with some exemplary patterns based on the identification of largest coefficients in rows. Let $\bar{S}_i^{\texttt{max-n}}(A, q_i)$ be the index set $\{j_1, \ldots, j_{q_i}\}$ of column indices in row i of matrix A, such that the $|A_{i,j_k}|$ are the q_i largest off-diagonal absolute values in this row. Then

$$S^{\texttt{max-n}}(A,q) := \{(i,j) \mid j \in \bar{S}_i^{\texttt{max-n}}(A, q_i)\} \tag{49}$$

$$S^{\texttt{max-t}}(A,\tau) := \{(i,j) \mid |A_{i,j}| \ge \tau_i \,\texttt{max}_k |A_{i,k}|\}\,, \tag{50}$$

parameterized by vectors q and τ give us different options for the row-specific selection of largest coefficients: q largest ($S^{\texttt{max-n}}(A,q)$), above threshold ($S^{\texttt{max-t}}(A,\tau)$). The diagonal is often included even if it has smaller coefficients, e.g. $S := S^{\texttt{diag}} \cup S^{\texttt{max-n}}(A,q)$.

Given a matrix A and a sparsity pattern S we want to be able to select and scale coefficients in A based on the sparsity pattern

$$(\texttt{prune}(A,S))_{i,j} := \begin{cases} A_{i,j} & \texttt{if } (i,j) \in S \\ 0 & \texttt{else} \end{cases} \tag{51}$$

$$(\texttt{scale}(A,S,\omega))_{i,j} := \begin{cases} \omega\, A_{i,j} & \texttt{if } (i,j) \in S \\ A_{i,j} & \texttt{else} \end{cases}\,. \tag{52}$$

After pruning we want to quantitatively measure the coefficient weights within the sparsity pattern in relation to the *matrix weight* of all coefficient $\|A\|_{1,1} := \sum_{i,j} |A_{i,j}|$. For

this purpose we define the S-*coverage* $\texttt{coverage}(A,S)$ with respect to a sparsity pattern S and the *diagonal coverage* $c_{\texttt{diag}}(A)$ as a special case

$$\texttt{coverage}(A,S) := \|\texttt{prune}(A,S)\|_{1,1}/\|A\|_{1,1} \ \in [0,1] \tag{53}$$

$$c_{\texttt{diag}}(A) := \texttt{coverage}(A,S^{\texttt{diag}}) = \|\texttt{diag}(A)\|_{1,1}/\|A\|_{1,1}\,. \tag{54}$$

In the original ordering of indices the sparsity pattern S of a matrix $B := \texttt{prune}(A,S)$ might appear irregular, but in a new ordering P (P is an index permutation) $\hat{B} := PBP^T$ could reveal a particular pattern $\hat{S}$, e.g. $\hat{S} \subseteq S^{\texttt{tri}}$ thus $\hat{B}$ being tridiagonal. We use the same notation for permutations acting on matrices and on sparsity patterns, i.e. if B has sparsity pattern S then $\hat{B}$ has sparsity pattern $\hat{S}$

$$\hat{B} := PBP^T\,, \quad \hat{S} := PSP^T\,; \quad B = P^T\hat{B}P\,, \quad S = P^T\hat{S}P \tag{55}$$

$$B = \texttt{prune}(A,S) = P^T\,\texttt{prune}(PAP^T, PSP^T)\,P\,. \tag{56}$$

The construction of S aims at a particular sparsity pattern $\hat{S} = PSP^T$ in the new ordering P, e.g. $\hat{S} \subseteq S^{\texttt{tri}}$ so that $\hat{B}$ is tridiagonal, but in general we cannot assume equality $\hat{S} \neq S^{\texttt{tri}}$, because the construction of S might have excluded certain coefficients of A.

5.1.9. Direct Construction

Starting with the MOS ansatz in Eq. 37 we define the factors M_l in dependence on some off-diagonal matrices $A''_0, \ldots, A''_{m-1}$ and an invertible diagonal matrix T with $\texttt{unit}(T) = \texttt{unit}(\texttt{diag}(A))$ and $|T| \geq |\texttt{diag}(A)|$

$$E := \frac{1}{m}T^{-1}(\texttt{diag}(A) - T) \leq 0, \quad T\,(I + mE) = \texttt{diag}(A) \tag{57}$$

$$J := I + E, \quad \frac{m-1}{m}I \leq J \leq I \tag{58}$$

$$M'_l := (TE + A'_l)\,, \quad A'_l := J^{-(m-1-l)}A''_l J^{-l} \tag{59}$$

$$M_l := (I + T^{-1}M'_l) = (J + T^{-1}A'_l)\,, \tag{60}$$

where the comparison, absolute value and $\texttt{unit}()$ (Eq. 43) operators act on the coefficients of the matrices. Eq. 58 and Bernoulli's inequality [79] bound the powers of J for all $m \geq 2$

$$I \leq J^{-m} \leq 4I, \quad 0 \leq J^m - (I + mE) \leq \binom{m}{2}E^2 \leq \frac{1}{2}(mE)^2\,. \tag{61}$$

We obtain the MOS-d preconditioner as a perturbation of A

$$M_{\text{MOS-d}} := T\prod_{l=m-1}^{0}(I + T^{-1}M'_l) = T\prod_{l=m-1}^{0}(J + T^{-1}A'_l) \tag{62}$$

$$= TJ^m + \sum_{l=0}^{m-1} J^{m-1-l}A'_l J^l + R_2$$

$$= A + R, \quad R := R_0 + R_1 + R_2 \tag{63}$$

$$R_2 := \sum_{l_1,l_2=0;\,l_1>l_2}^{m-1} A''_{l_1}J^{-m}T^{-1}A''_{l_2} + R_{\text{tail}}(\|T^{-1}\|^2)$$

$$R_1 := \left(\texttt{diag}(A) + \sum_{l=0}^{m-1}A''_l\right) - A$$

$$R_0 := T\,(J^m - (I + mE))\,, \quad \|R_0\| \leq \frac{1}{2}\|T\|\|mE\|^2\,.$$

From matrix perturbation theory [119] we know a sufficient condition for invertibility

$$\|A^{-1}\|\|R\| < 1 \quad \Longleftrightarrow \quad \frac{\|R\|}{\|A\|} < \frac{1}{\kappa(A)} \tag{64}$$

and quantitative bounds in an induced matrix norm $\|.\|$

$$\frac{\|A^{-1} - M_{\text{MOS-d}}^{-1}\|}{\|A^{-1}\|} \leq \|I - M_{\text{MOS-d}}^{-1}A\| \leq \frac{\|A^{-1}\|\|R\|}{1 - \|A^{-1}\|\|R\|} \,. \tag{65}$$

For a given A we can check if $\|R\|/\|A\|$ is sufficiently small, but estimating $\kappa(A)$ and computing $R = M_{\text{MOS-d}} - A$ is expensive as it involves $m-1$ sparse matrix-matrix products. It will require further investigation to determine more economic criteria.

We can force $R_1 = 0$ by distributing all off-diagonal coefficients of A onto the A_l'' in any way we like. However, if A has many off-diagonal coefficients this would require either many A_l'' with few coefficients (many simple M_l inversions) or few A_l'' with many coefficients (few difficult M_l inversions), so for matrices with many off-diagonal elements forcing $R_1 = 0$ requires too much effort. Instead, we distribute only selected off-diagonal coefficients of A onto the A_l'', i.e. we enforce

$$\texttt{prune}(A, S_0) = \left(\texttt{diag}(A) + \sum_{l=0}^{m-1} A_l''\right) , \quad \|R_1\| = \|\texttt{prune}(A, S_0) - A\| \tag{66}$$

for some application specific sparsity pattern S_0, e.g. $S_0 := S^{\texttt{diag}} \cup S^{\texttt{max-t}}(A, \tau)$ (Eq. 50). In this general setting of Section 5.1 this intentionally leaves many options for choosing m and A_l'', and thus for adapting to the properties of A.

$\|R_0\|$ is bounded by $\frac{1}{2}\|T\|\|T^{-1}(\texttt{diag}(A) - T)\|^2$ so by choosing T sufficiently close to $\texttt{diag}(A)$ we can make the error arbitrary small. $\|R_2\|$ is bounded by $c(A)\|T^{-1}\|$ so by choosing $|T|$ sufficiently large we can make the error arbitrary small. Hence, the separate consideration of $\|R_0\|$ and $\|R_2\|$ leads to contradicting requirements on T. In fact, we should simultaneously minimize $\|R_0 + R_2\|$ for better error control, but this is not so easy. The first order term in R_2 does not contribute to the diagonal if the symmetrized sparsity patterns of all A_l'' are disjoint. So in these cases only the higher order terms in R_2 interact with the diagonal R_0.

There are more refined choices for T, but in favor of a faster construction of $M_{\text{MOS-d}}$, here we choose a simple compromise between the contradicting requirements

$$T := \texttt{diagp}_1(A) \,. \tag{67}$$

If the resulting $R = M_{\text{MOS-d}} - A$ (Eq. 63) satisfies the sufficient condition $\|R\|/\|A\| < 1/\kappa(A)$ then Eq. 65 holds, else properties of $M_{\text{MOS-d}}$ must be checked explicitly.

5.1.10. Adaptive Construction

We begin with the MOS ansatz (Eq. 37)

$$M_{\text{MOS-a}} := T \prod_{l=m-1}^{0} M_l \,, \tag{68}$$

but now the M_l are constructed recursively with intermediate pruning (Eq. 51)

$$\begin{aligned} B_0 &:= \texttt{prune}(A, S_0^B) \quad M_0 := \texttt{prune}(B_0, S_0^M) \\ B_1 &:= \texttt{prune}(B_0 M_0^{-1}, S_1^B) \quad M_1 := \texttt{prune}(B_1, S_1^M) \\ &\vdots \\ B_{m-1} &:= \texttt{prune}(B_{m-2} M_{m-2}^{-1}, S_{m-1}^B) \quad M_{m-1} := \texttt{prune}(B_{m-1}, S_{m-1}^M) \\ B_m &:= \texttt{prune}(B_{m-1} M_{m-1}^{-1}, S_m^B) \quad T := \texttt{diagp}_1(B_m) \\ & S_l^B \supseteq S_l^M \,. \end{aligned} \tag{69}$$

If B_l or M_l become singular, they can always be fixed a-posteriori by changing S_l^B and S_l^M, but repeatedly checking for invertibility is expensive. So far we do not have any S_l^B and S_l^M with a-priori guarantees of invertibility in case of a general A.

If we did not prune any B_l then we would have $B_l = B_{l-1} M_{l-1}^{-1} = A M_0^{-1} \dots M_{l-1}^{-1}$, i.e. the so far constructed preconditioner applied to A and thus a very good measure of where the preconditioner is still lacking. However, the resulting B_l would be dense and too big to store, so they are pruned with S_l^B already during the computation, i.e. $B_l = \texttt{prune}(B_{l-1} M_{l-1}^{-1}, S_l^B)$ is an incomplete sparse inversion of M_{l-1} applied to B_{l-1}. Consequently, the B_l are only approximations of the so far constructed preconditioner applied to A. The accuracy of the B_l can be observed by computing $\|A - B_l \prod_{k=l-1}^{0} M_k\|$ but this involves sparse matrix-matrix products.

In choosing S_l^B the user enters a tradeoff between the accuracy of the B_l and their construction time, e.g. if $S_l^B := S^{\texttt{diag}} \cup S^{\texttt{max-n}}(B_{l-1} M_{l-1}^{-1}, q)$ (Eq. 49) then q controls the number of output non-zeros per row in each B_l and thus also the number of input non-zeros for $l + 1$; with a known input and output size the execution time can be estimated.

In choosing S_l^M the user enters a tradeoff between many M_l with few coefficients (many simple M_l inversions) or few M_l with many coefficients (few difficult M_l inversions) during the preconditioning with $M_{\text{MOS-a}}$ (Eq. 68). Which one is faster depends on properties of A, of the parallel algorithms and of the available hardware. In this general setting of Section 5.1 the above construction (Eq. 69) intentionally leaves many options for choosing m, S_l^B and S_l^M and thus for adapting to the properties of the matrix A.

5.2. Tridiagonal Splittings

This section examines the special case of tridiagonal splittings. As we operate in a purely algebraic setting with only the matrix A, this is much more involved than ADI methods on a structured grid. The alternating splittings $\overline{M}_l$ used by the $\overline{M}_{\text{ALT-o}}$ and $\overline{M}_{\text{ALT-i}}$ preconditioners and the matrix factors M_l of the MOS ansatz in Eq. 32 used by the $M_{\text{MOS-d}}$ and $M_{\text{MOS-a}}$ preconditioners will now be tridiagonal in some associated ordering P_l (cf. Section 5.1.8).

In Chapter 4 we presented the extraction of a maximum linear forest from a weighted graph and the computation of the corresponding permutation. Then the first challenge is to decompose the adjacency graph of A into multiple (almost) disjoint linear forests by an efficient iterative process and thus obtain all the P_l (Section 5.2.1). The second challenge is to compute the values in the M_l (Sections 5.2.2 and 5.2.3), setting the $\overline{M}_l$ values is easy (Section 5.2.1). The third challenge is to handle the permutations during execution (Section 5.2.4).

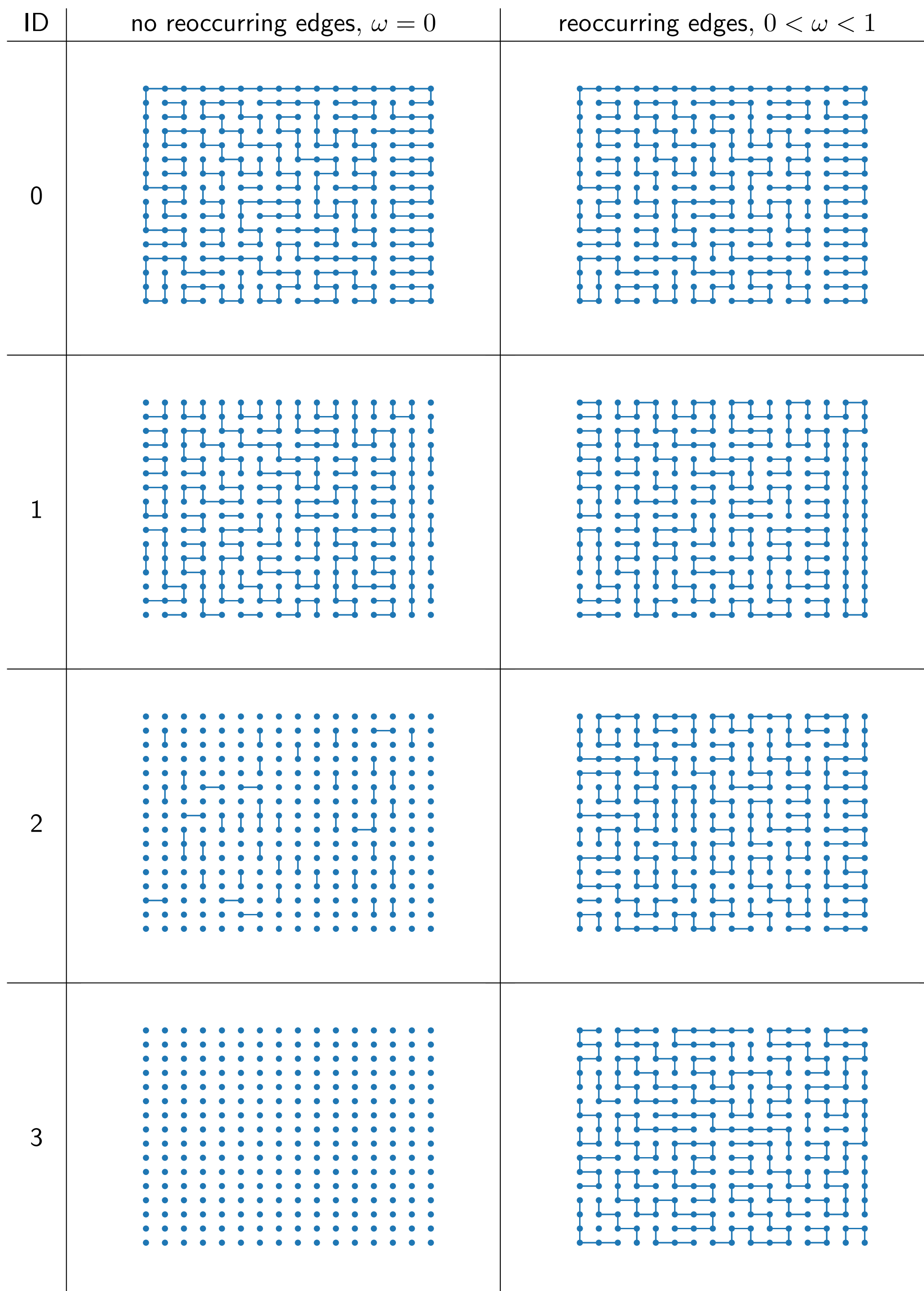


Figure 33.: Subsequent segmentations (top to bottom) of a 2D isotropic grid (5pt stencil) without reoccuring edges ($\omega = 0$) in the left and reoccuring edges ($0 < \omega < 1$) in the right column.

5.2.1. Iterative Decomposition into Linear Forests

The segmentation does not use the original matrix A but rather a normalized and symmetrized version of it

$$C_0 := |\texttt{diagp}_1(A)^{-1}A| + |\texttt{diagp}_1(A)^{-1}A|^T \,. \tag{70}$$

Firstly, this ensures invariance under diagonal scaling (Section 5.1.7). Secondly, this helps our symmetric second phase with the edge confirmation: if an edge has a large weight in one direction and thus should be included in the segmentation, the edge in the other direction will be likely proposed too.

Executing the segmentation and permutation (Chapter 4) on C_0 delivers the first sparsity pattern S_0 and the first permutation P_0. For the following S_l, P_l the same procedure is repeated but in every iteration l we use a different auxiliary graph C_l, because we want to extract a different linear forest. Therefore, before each new iteration l we perform a *weight adjustment*

$$C_l := \texttt{scale}(C_{l-1}, S_{l-1} \setminus S^{\texttt{diag}}, \omega) \tag{71}$$

with the scaling function from Eq. 52. This has the effect, that previously selected edges are either not chosen again ($\omega = 0$, default for $M_{\text{MOS-d}}$) or chosen less likely ($0 < \omega < 1$, default for $\overline{M}_{\text{ALT-o}}$ and $\overline{M}_{\text{ALT-i}}$). The effect of ω is shown in Figure 33 for a small exemplaric graph of a 5pt stencil. For $M_{\text{MOS-a}}$ the C_l have a different adaptive construction (Eq. 75) and we refer to Section 5.2.3.

From S_l, P_l we can immediately compute the tridiagonal alternating splittings (cf. the example Eq. 56)

$$\overline{M}_l := \texttt{prune}(A, S_l) = P_l^T \,\texttt{prune}(P_l A P_l^T, P_l S_l P_l^T)\, P_l \tag{72}$$

where $P_l S_l P_l^T \subseteq S^{\texttt{tri}}$. In other words, the tridiagonal alternating splittings $\overline{M}_l$ simply extract coefficients from A. In contrast, the coefficients of the matrix factors M_l in Eq. 32 contain different coefficients as explained in the following subsections.

5.2.2. Direct Tridiagonal Construction

We follow the general construction in Section 5.1.9 to obtain the preconditioner $M_{\text{MOS-d}}$ (Eq. 62) with matrix factors M_l (Eq. 60), which are tridiagonal in the ordering of P_l.

All segmentations S_l and permutations P_l are computed from the helper matrices C_l (Eqs. 70 and 71) according to the algorithms in Chapter 4. We set $T := \texttt{diagp}_1(A)$ (Eq. 67) and

$$A''_l := \texttt{prune}(A, S_l) = P_l^T \,\texttt{prune}(P_l A P_l^T, P_l S_l P_l^T)\, P_l \tag{73}$$

where $P_l S_l P_l^T \subseteq S^{\texttt{tri}}$. The A''_l have symmetric sparsity patterns which are disjoint, because S_l is symmetric and by default a zero weight adjustment factor is used (Eq. 71). We have observed that $M_{\text{MOS-d}}$ performs better with this setting.

The M_l are calculated in multiple steps (Eqs. 57-60) from T and A''_l, they are tridiagonal in the ordering of P_l. Therefore, the inversion of $M_{\text{MOS-d}}$ (Eq. 62) is a series of successive permutations and tridiagonal solves, see Section 5.2.4 for more details.

A heuristic for m is to stop growing l at $\texttt{coverage}(A, \cup_{k=0}^{l} S_k) \geq \mu$ or at onset of stagnation in the increase, i.e. when the A''_l cover a certain fraction of the matrix weight $\|A\|_{1,1}$ or the coverage stagnates.

5.2.3. Adaptive Tridiagonal Construction

We follow the general construction in Section 5.1.10 to obtain the preconditioner $M_{\text{MOS-a}}$ (Eq. 68) with matrix factors M_l (Eq. 69), which are tridiagonal in the ordering of P_l.

We list the dependencies and the order of computation for the entities from Eq. 69 and the helper matrices C_l used for the segmentations S_l^M and the permutations P_l, starting with $l = 0, B_{-1} := A,\ M_{-1} := I$

$$\begin{aligned} &S_l^B(B_{l-1}, M_{l-1})\,, \quad B_l(B_{l-1}, M_{l-1}, S_l^B)\,, \\ &C_l(B_l)\,, \quad S_l^M(C_l)\,, \quad P_l(S_l^M)\,, \quad M_l(B_l, S_l^M, P_l)\,. \end{aligned} \tag{74}$$

In particular, in contrast to Section 5.2.2, where all C_l depend only on A (Eq. 71) and can all be computed before the M_l computations, here the C_l implicitly depend on previous M_l and therefore the M_l computations are intermingled with the segmentation and permutation.

For a specific l, we now specify the construction of those entities in the above dependency order. Let q_i be the number of off-diagonal non-zeros in row i of A, then staring with $S_0^B := A$

$$S_l^B := S^{\texttt{diag}} \cup S^{\texttt{max-n}}(B_{l-1}M_{l-1}^{-1}, q)\,, \quad B_l := \texttt{prune}(B_{l-1}M_{l-1}^{-1}, S_l^B)\,,$$

with $S^{\texttt{max-n}}$ from Eq. 49. In other words, in the first step we do not prune at all $B_0 = A$ and in the following steps we keep the largest non-zeros in each row, such that B_l and A have the same number of non-zeros in each row, but in varying column positions depending on where the largest coefficients in $B_{l-1}M_{l-1}^{-1}$ lie.

The S_l^M and P_l are computed from the helper matrix

$$C_l := |\texttt{diagp}_1(B_l)^{-1}B_l| + |\texttt{diagp}_1(B_l)^{-1}B_l|^T \tag{75}$$

according to the algorithms in Chapter 4. Finally, we obtain the adaptive matrix factors M_l which are tridiagonal in the ordering of P_l because $P_l S_l^M P_l^T \subseteq S^{\texttt{tri}}$

$$M_l := \texttt{prune}(B_l, S_l^M) = P_l^T\, \texttt{prune}(P_l B_l P_l^T, P_l S_l^M P_l^T)\, P_l\,. \tag{76}$$

The adaptive construction works well, if the B_l are good approximations of the so far constructed preconditioner applied to A, and become increasingly diagonally dominant with growing l, which shows in the increase of the diagonal coverage $c_{\texttt{diag}}(B_l)$ (Eq. 54). We have observed this favorable behavior for many matrices, however, for some isotropic matrices with a high mean degree $c_{\texttt{diag}}(B_l)$ does not increase. A heuristic for m is then to stop growing l at $c_{\texttt{diag}}(B_l) \geq \mu$ or at onset of stagnation in the increase.

5.2.4. Preconditioning with Fused Permutations

While many permutations P_l appear in previous formulas, they do not imply data movement. This subsection describes the few places where data is actually permuted in memory. The employed tridiagonal solver works with a banded matrix format, so each matrix factor M_l is stored as a tridiagonal system A_l in three bands together with a permutation P_l

$$A_l := P_l M_l P_l^T\,, \quad M_l = P_l^T A_l P_l\,. \tag{77}$$

The iterations themselves (Eqs. 34 and 35) operate on the original ordering, therefore, the application of M_l^{-1} to a vector r requires a permutation before and after

$$M_l^{-1} r = P_l^T\ A_l^{-1}(P_l r)\,. \tag{78}$$

As the multiplicative ansatz for M_{MOS} (Eq. 32) employs the inversions M_l^{-1} successively we have

$$M_{\text{MOS}}^{-1} r = P_0^T A_0^{-1} P_0 \, P_1^T \cdots P_{m-2} \, P_{m-1}^T A_{m-1}^{-1} P_{m-1} T^{-1} \, r \,. \tag{79}$$

Each permutation pair $P_l P_{l+1}^T$ is fused and implemented as a single permutation. Thus for m tridiagonal systems, only $m + 1$ vector permutations are executed.

The handling of the banded storage format (Eq. 77) and an individual inversion (Eq. 78) applies equally to the alternating splittings $\overline{M}_l$ as to the matrix factors M_l. But the product inversion (Eq. 79) does not apply to $\overline{M}_l$, because they are always used individually on vectors (Eqs. 28 and 35).

5.3. Results

All measurements presented in this thesis were done on a machine with CentOS 7, CUDA Toolkit 11.2.142, CUDA driver 450.57, host compiler GCC 10.2.0, and a GeForce RTX 2080 Ti.

5.3.1. Overview of Solvers

Overall, we evaluate our four approaches with alternating and multiplicative tridiagonal operator splittings and the diagonal preconditioners for general sparse matrices. Enumerated in the order of decreasing execution complexity for one k-iteration these are:

1. `FGMRES`($\overline{M}_{\text{ALT-o}}$): Eq. 35 with outer alternation of alternating splittings $\overline{M}_l$ from Eq. 29,
2. `GMRES`($\overline{M}_{\text{ALT-i}}$): Eq. 34 with inner alternation (applying $\overline{M}_{\text{ALT-i}}$ from Eq. 30 recursively with Eq. 28) of alternating splittings $\overline{M}_l$ from Eq. 29,
3. `GMRES`($M_{\text{MOS-d}}$): Eq. 34 with direct MOS (Eq. 62) comprised of matrix factors M_l from Eq. 60 and T from Eq. 67,
4. `GMRES`($M_{\text{MOS-a}}$): Eq. 34 with adaptive MOS (Eq. 68) comprised of matrix factors M_l and T from Eq. 69,
5. `GMRES(diagp`$^m(A)$`)`: Eq. 34 with diagonal preconditioners from Eq. 39, 40 and 41.

All solvers run in parallel on the GPU and are evaluated on numerous challenging problems listed in Table 17.

5.3.2. General Settings

The self-generated matrices ANISO1 and ANISO2 are based on anisotropic 9pt stencils in 2D:

$$\text{ANISO1} = \begin{pmatrix} -0.2 & -0.1 & -0.2 \\ -1.0 & 3.0 & -1.0 \\ -0.2 & -0.1 & -0.2 \end{pmatrix}, \ \text{ANISO2} = \begin{pmatrix} -0.1 & -0.2 & -1.0 \\ -0.2 & 3.0 & -0.2 \\ -1.0 & -0.2 & -0.1 \end{pmatrix}.$$

The solution of the equation system is always initialized with $\hat{x}_i := \sin(2\pi f i / N)$, where N denotes the degrees of freedom (DOFs) of the corresponding problem and $f := 8$. Subsequently, the right-hand side is calculated from this solution vector. Therefore, we are able to compute the forward relative error of the solver

$$\text{FRE} := \|x^{k_{\text{last}}} - \hat{x}\|_2 / \|\hat{x}\|_2, \tag{80}$$

Name	Problem	Origin	DOFs	nnz	$\bar{n}_r$	$c_{\texttt{diag}}$	$\hat{n}_r$
MODEL2	27pt isotropic Laplacian	hypre	1 000 000	26 463 592	26.46	0.51	27
ATMOSMODL	fluid dynamics	SMC	1 489 752	10 319 760	6.93	0.50	7
MODEL8	5pt isotropic	hypre	1 000 000	4 996 000	5.00	0.50	5
ANISO1	9pt 2D stencil	A	1 440 000	12 945 604	8.99	0.50	9
ANISO2	9pt 2D stencil	A	1 440 000	12 945 604	8.99	0.50	9
G3_CIRCUIT	circuit simulation	SMC	1 585 478	7 660 826	4.83	0.76	6
GEO_1438	structural	SMC	1 437 960	63 156 690	43.92	0.33	57
MODEL4	rotated 2D anisotropic	hypre	1 000 000	6 992 002	6.99	0.39	7
MODEL3	rotated 2D anisotropic	MFEM	1 002 001	9 006 001	8.99	0.44	9
MODEL5	regions with anisotropies	MFEM	1 002 001	9 006 001	8.99	0.45	9
MODEL7	grid aligned anisotropic	MFEM	1 002 001	9 006 001	8.99	0.33	9
ECOLOGY2	2D/3D	SMC	999 999	4 995 991	5.00	0.50	5
ML_GEER	structural	SMC	1 504 002	110 879 972	73.72	0.22	74
TRANSPORT	structural	SMC	1 602 111	23 500 731	14.67	0.50	15
THERMAL2	thermal	SMC	1 228 045	8 580 313	6.99	0.50	11
HOOK_1498	structural	SMC	1 498 023	60 917 445	40.67	0.31	93
AF_SHELL10	structural	SMC	1 508 065	52 672 325	34.93	0.39	35

Table 17.: Matrices from the Sparse Matrix Collection [27] (SMC), generated with MFEM [6], hypre [45], and self-created (A); horizontal grouping corresponds to Figures 36, 37, 38; $c_{\texttt{diag}}$ is the diagonal weight coverage of the matrix (Eq. 54), $\bar{n}_r$ the mean and $\hat{n}_r$ the maximum number of non-zeros per row.

from the solution $\hat{x}$ and last iterate $x^{k_{\text{last}}}$. Please note that, contrary to the residual norm, the FRE does not neccessarily decrease monotonically for GMRES. Moreover, for problems with a high condition number, FRE can be large even when the solver has successfully converged to a small relative residual norm.

For the benchmarks, our preconditioners run in combination with the MAGMA library [8], which also provides the implementations for GMRES, ILU(0) and ILU(0)-ISAI(m) preconditioner by Anzt et al. [7]. The ILU(0)-ISAI(m) solves the triangular factors in ILU(0) with a stationary iteration in m relaxation steps, e.g., instead of the exact inversion $y = L^{-1}z$ with the lower triangular factor L we compute for $s = 0, \dots, m-1$, $M_L \approx L^{-1}$

$$y^0 := z, \quad y^{s+1} := M_L(z - Ly^s),$$

such that $y^m \approx y = L^{-1}z$. For some matrices we do not have ILU-ISAI results because the ILU-ISAI implementation in MAGMA is restricted by the amount of non-zero elements per row of the sparse matrix. The restart parameter of the GMRES is set to 20. The twenty GMRES steps with preconditioning execute in single precision on the GPU, an outer mixed precision iterative refinement [54] ensures more precise computation of the residual in double precision.

The preconditioners $\overline{M}_{\text{ALT-o}}$ and $\overline{M}_{\text{ALT-i}}$ use a weight adjusting factor of $\omega = 0.01$ (Eq. 71) in segmentations, the preconditioner $M_{\text{MOS-d}}$ uses $\omega = 0$ by default, the other preconditioners do not use this parameter.

Our preconditioners solve the tridiagonal systems with our tridiagonal solver [72] which fully utilizes the GPU memory bandwidth for large systems. Therefore, exposing more parallelism in the preconditioner application, for example using additive operator splitting schemes, would only achieve a higher level of hardware utilization for small problem sizes.

5.3.3. Discussion

The convergence results are presented in three groups based on the number of iterations required to decrease the relative residual norm below 10^{-6}. For each group we show the numerical convergence and the execution time without setup.

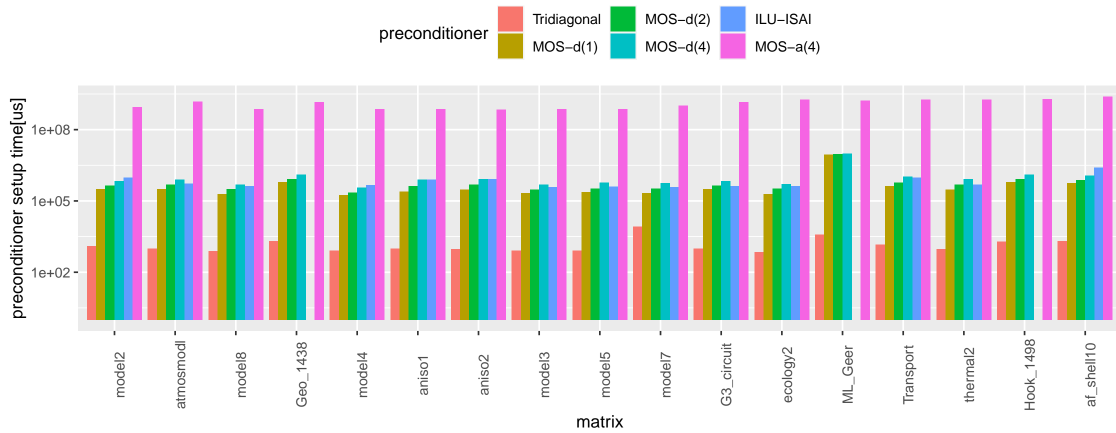


Figure 34.: Preconditioner setup times for each test matrix from Table 17. For matrix ML_GEER, the GPU memory was oversubscribed during the setup of MOS-d, which resulted in page evictions and migrations of CUDA managed memory, and thus significant higher setup times than for the other matrices.

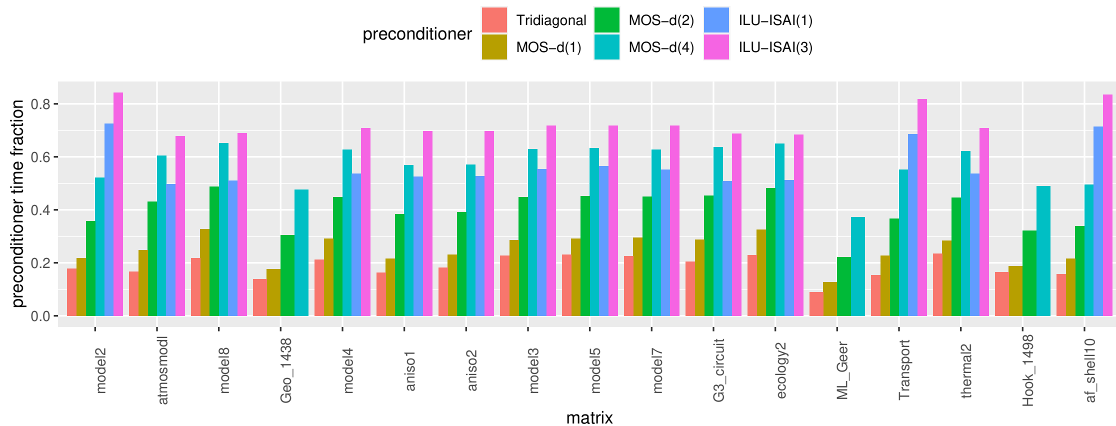


Figure 35.: Relative time of the preconditioner for one GMRES iteration (single precision).

The top halfs of Figures 36, 37, 38 present the numerical convergence results for x^k with respect to k, the number of GMRES iterations in Eq. 34, except for scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) which performs m iterations for each k. The bottom halfs of the figures show numerical convergence with respect to time per non-zero. We divide the time by the number of non-zeros of the corresponding matrix, so that execution effort for different matrices is easier to compare.

Five columns correspond to our five schemes from Section 5.3.1, the fourth column uses GMRES in combination with ILU-ISAI. Within the same plot the differently colored curves show how the convergence changes with m. For the preconditioners, which depend on m, the corresponding preconditioner runtime increases linearly with m. Therefore, the numerical convergence improvement in terms of fewer iterations must compensate for the additional runtime of the preconditioner.

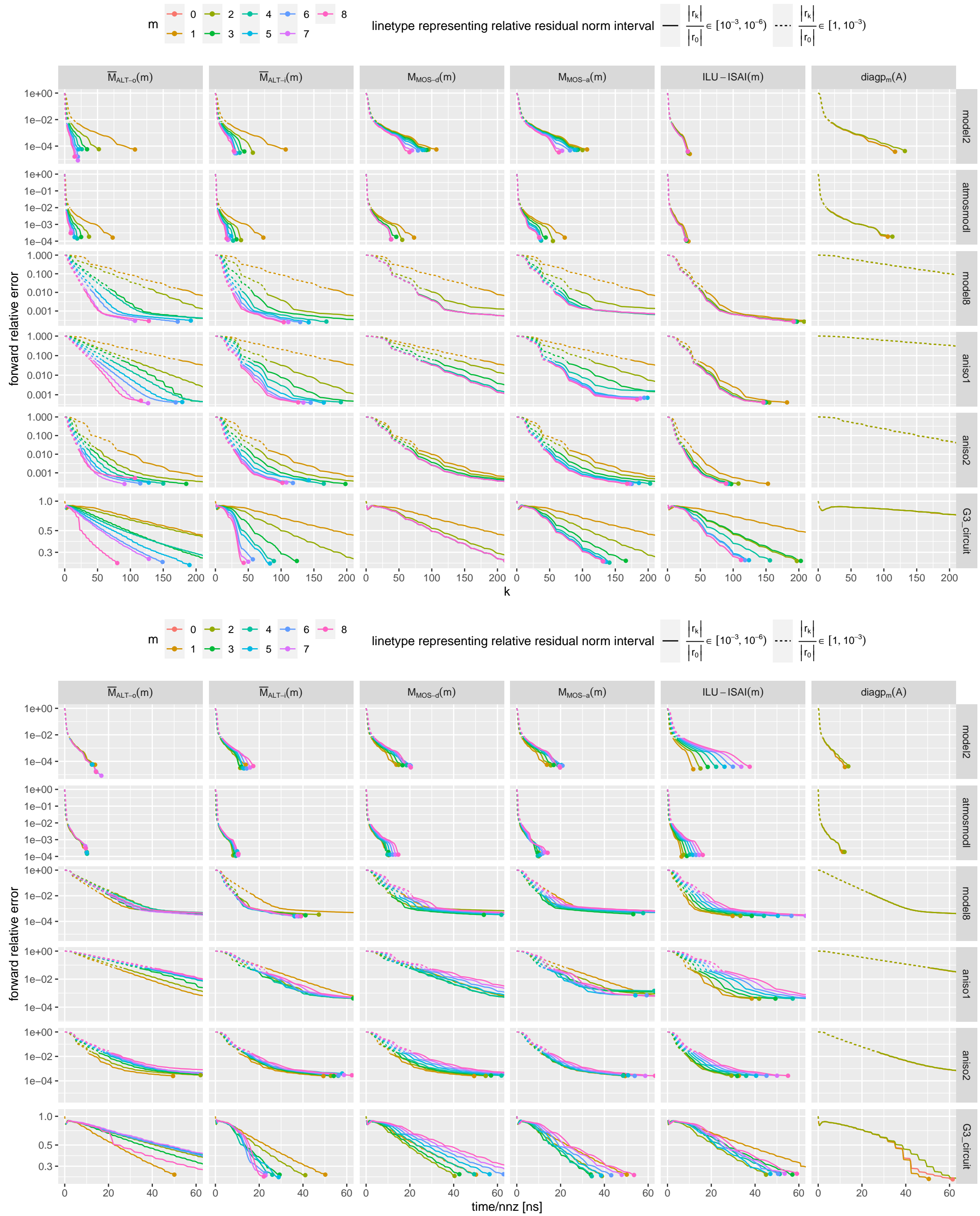


Figure 36.: GMRES relative forward error on logarithmic y-scale against number of iterations (top) and against time per non-zero (bottom). Setup times are not included. Where the curves end with a dot, the corresponding relative residual norm has fallen below 10^{-6}. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. The first 4 preconditioners are identical for $m = 1$, this common curve helps to compare them among each other.

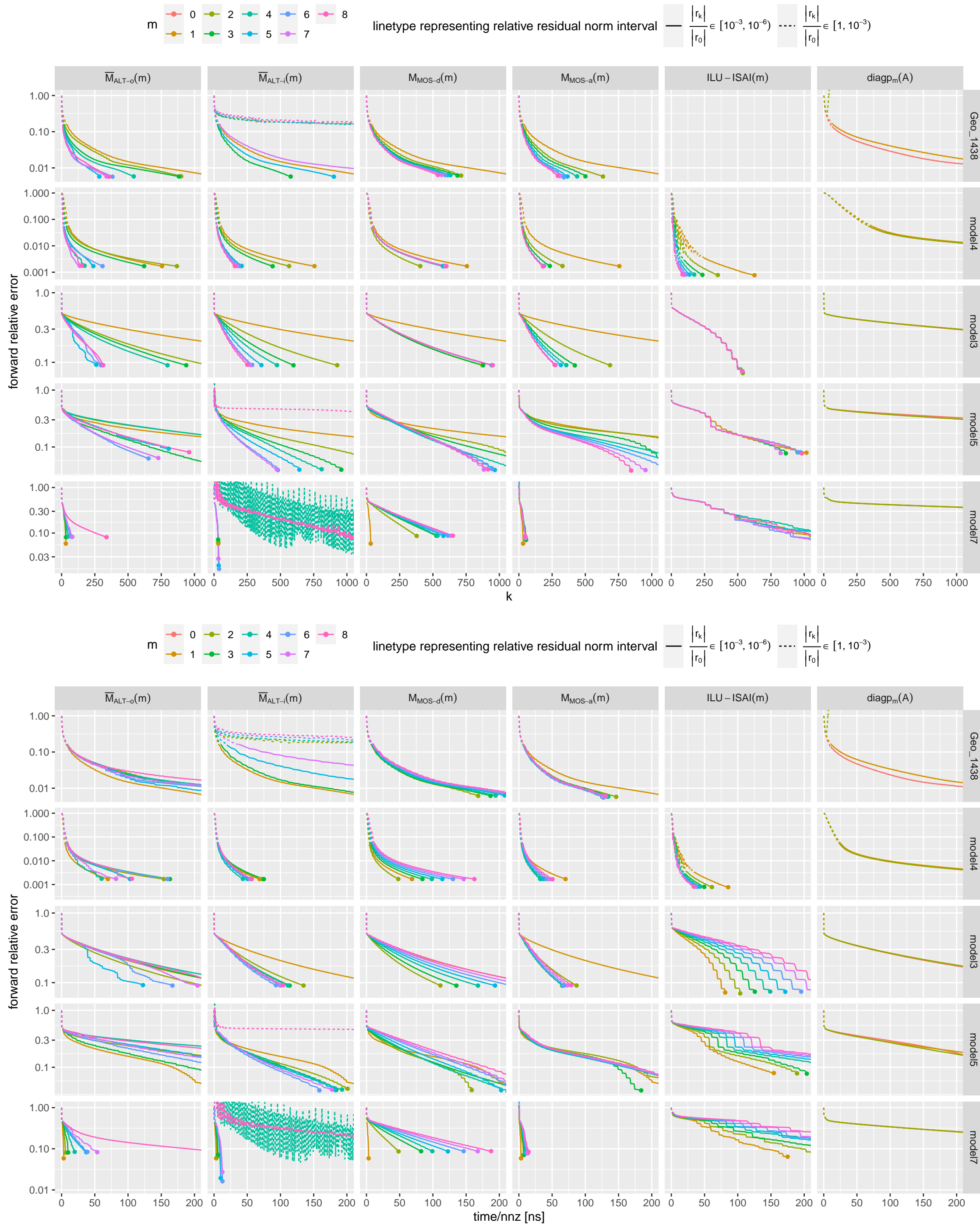


Figure 37.: GMRES relative forward error on logarithmic y-scale against number of iterations (top) and against time per non-zero (bottom). Setup times are not included. Where the curves end with a dot, the corresponding relative residual norm has fallen below 10^{-6}. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. The first 4 preconditioners are identical for $m = 1$, this common curve helps to compare them among each other.

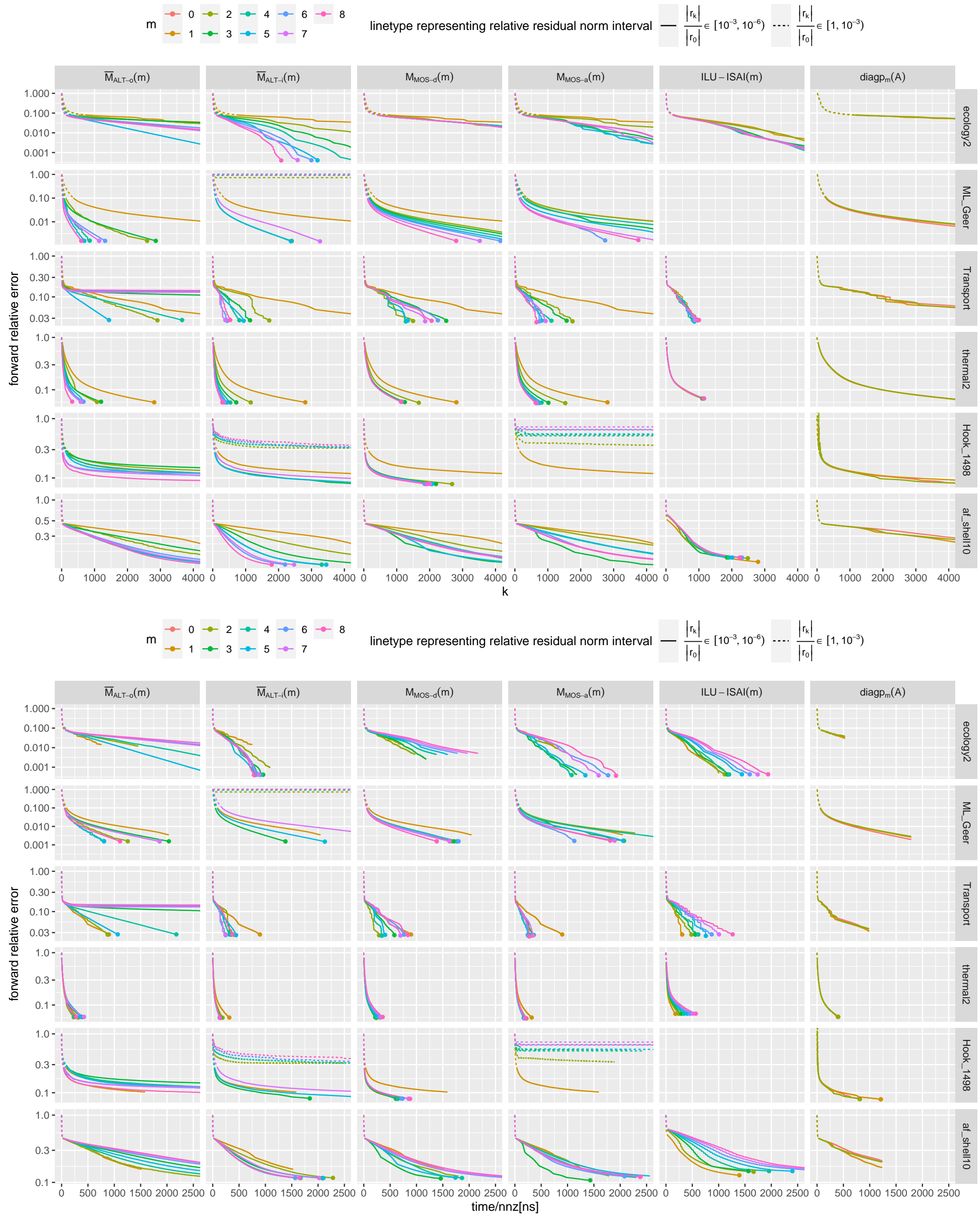


Figure 38.: GMRES relative forward error on logarithmic y-scale against number of iterations (top) and against time per non-zero (bottom). Setup times are not included. Where the curves end with a dot, the corresponding relative residual norm has fallen below 10^{-6}. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k. The first 4 preconditioners are identical for $m = 1$, this common curve helps to compare them among each other.

Diagonally Dominant Problems

The first group (Figure 36) has the easiest problems to solve. Overall $\overline{M}_{\text{ALT-i}}$ has the best convergence, but ILU runs fastest with the exception of G3_CIRCUIT, for which $\overline{M}_{\text{ALT-i}}$ is faster.

Matrix G3_CIRCUIT is interesting because despite strong diagonal dominance the condition number is high, so FRE remains high even though the relative residual norm quickly falls below 10^{-6}. All operator splitting preconditioners show a monotonic improvement in convergence per iteration with an increase in m. However, most matrices here have few non-zeros per row, so once all off-diagonal coefficients are part of some splitting bigger m have little to gain, most visible for MODEL8 and $m > 2$. So this group is quite unsuitable for our preconditioners, yet their performance keeps up with ILU reasonably well.

Non-Diagonally Dominant Problems

The second group (Figure 37) contains more challenging matrices with low diagonal coverage $c_{\texttt{diag}}$, e.g. MODEL3,5,7 the three anisotropic finite element problems. $M_{\text{MOS-a}}$ represents the best choice for this group, $\overline{M}_{\text{ALT-o}}$ and $\overline{M}_{\text{ALT-i}}$ follow closely. $M_{\text{MOS-d}}$ struggles more and ILU has problems.

Apart from MODEL4 ILU does not perform well and does not show any improvement with increasing sweeps m. On MODEL4 ILU is actually fastest probably exploiting the few non-zeros per row, although there is an oscillating behaviour of the FRE during the first iterations. For matrix MODEL4, the schemes $\overline{M}_{\text{ALT-o}}$ and $\overline{M}_{\text{ALT-i}}$ show a significant improvement between $m = 3$ and $m = 4$, as $m = 4$ includes a second tridiagonal matrix with the strong coefficients. In $M_{\text{MOS-d}}$ edges occur at most once so convergence hardly improves for $m > 2$ whereas $M_{\text{MOS-a}}$ benefits from the additional factors in the adaptive construction.

Matrix MODEL7 represents an edge case for $M_{\text{MOS-a}}$ because the B_l converge towards the identity matrix during construction, but the spectral radius increases for $m > 1$ and then decreases again relative to $M_{\text{MOS-a}}(2)$.

For matrix GEO_1438, the diagonal preconditioner $\texttt{diagp}_2$ has mostly a forward relative error greater one in the first 1000 iterations.

Difficult Problems

The third group (Figure 38) presents the hardest problems to solve, they require most iterations to reduces the relative residual norm. In this group $\overline{M}_{\text{ALT-i}}$ is the clear winner, $M_{\text{MOS-a}}$ and ILU keep up on some problems but trail on others.

On ECOLOGY2 $\overline{M}_{\text{ALT-i}}$ performs far superior because the problem is difficult but the residual computation is rather quick due to few non-zeros per row. Matrix HOOK_1498 is surprising, the simple $M_{\text{MOS-d}}$ does better with more factors than the adaptive $M_{\text{MOS-a}}$. Another interesting case is ML_GEER. With such low diagonal coverage $c_{\texttt{diag}}$ we would not expect $\texttt{diagp}_1$ to perform almost as good as the more sophisticated preconditioners.

Relative Preconditioner Time

Figure 35 shows the runtime of different preconditioners relative to one diagonally preconditioned GMRES iteration. This relation is important because the speedup from a faster preconditioner does not proportionally translate into an overall speedup including the outer solver. The preconditioner labelled 'Tridiagonal' is the tridiagonal part of A in the original ordering. Therefore, no segmentation has been computed and no permutations

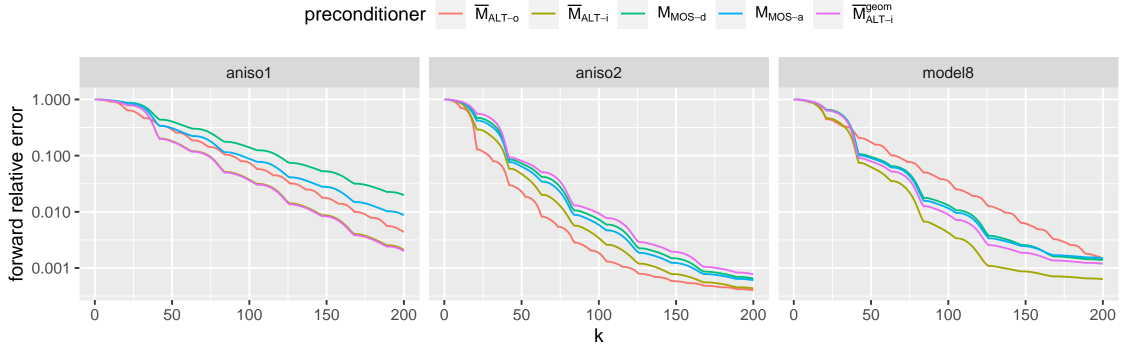


Figure 39.: GMRES relative forward error on logarithmic y-scale against number of iterations for operator split preconditioners with $m = 2$ in comparison to geometric ADI preconditioner $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ on 2D problems. Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k.

during its application are required. We see that the MOS-a(m) schemes execute faster than ILU-ISAI(1) for small m. Only MOS-a(4) requires more time per iteration than ILU-ISAI(1). The simple Tridiagonal preconditioner executes a little faster than $M_{\text{MOS-a}}(1)$ because no permutations during its execution are required.

Setup Time

Figure 34 shows setup times of different preconditioners, where $\overline{M}_{\text{ALT-o}}$ and $\overline{M}_{\text{ALT-i}}$ use approximately the same time as $M_{\text{MOS-d}}$. All setups run on the GPU. Setup of $M_{\text{MOS-a}}$ is costly because of the N^2 complexity. This could run much faster if we were dropping elements early as ILU does, but here we were focused on the strength of the preconditioner. The setup time for the operator split preconditioners increases linearly with m.

5.3.4. Comparison with Classical ADI Methods

Classical ADI preconditioners on tensor product grids are executed like our $\overline{M}_{\text{ALT-i}}$ preconditioner (Eq. 28), with the difference that the alternating $\overline{M}_l$ (Eq. 29) correspond to the different grid dimensions and can therefore be immediately extracted from A. We denote the resulting preconditioner $\overline{M}^{\text{geom}}_{\text{ALT-i}}$.

We compare our operator split preconditioners with $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ on two dimensional problems, thus, the $\overline{M}_0, \overline{M}_1$ splittings of $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ refer to the tridiagonal systems along the x-lines and y-lines in the grid, respectively. The convergence results are shown in Figure 39. For ANISO1 from Section 5.3.2, the first splitting of the geometric and algebraic preconditioners include the strong coefficients along the x-dimension, and thus, $\overline{M}_{\text{ALT-i}}$ and $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ perform equally well. For ANISO2, the $\overline{M}_{\text{ALT-i}}$ includes the strong coefficients of the stencil diagonal, but $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ does not and it shows the worst convergence of all schemes. Matrix MODEL8 is a homogeneous 5pt stencil. An example of the segmentation on a small graph induced by a 5pt stencil is shown in Figure 33. The geometric $\overline{M}^{\text{geom}}_{\text{ALT-i}}$ has the x-coefficients in $\overline{M}_0$ and the y-coefficients in $\overline{M}_1$. The algebraic $\overline{M}_{\text{ALT-i}}$ ($\omega = 0.01$) mixes the x- and y-coefficients in both of its splittings $\overline{M}_0, \overline{M}_1$ (see Figure 33, right column, both first rows). We see that the mixing works better than the separation in $\overline{M}^{\text{geom}}_{\text{ALT-i}}$.

In summary, the algebraic $\overline{M}_{\text{ALT-i}}$ is clearly superior to the geometric $\overline{M}^{\text{geom}}_{\text{ALT-i}}$.

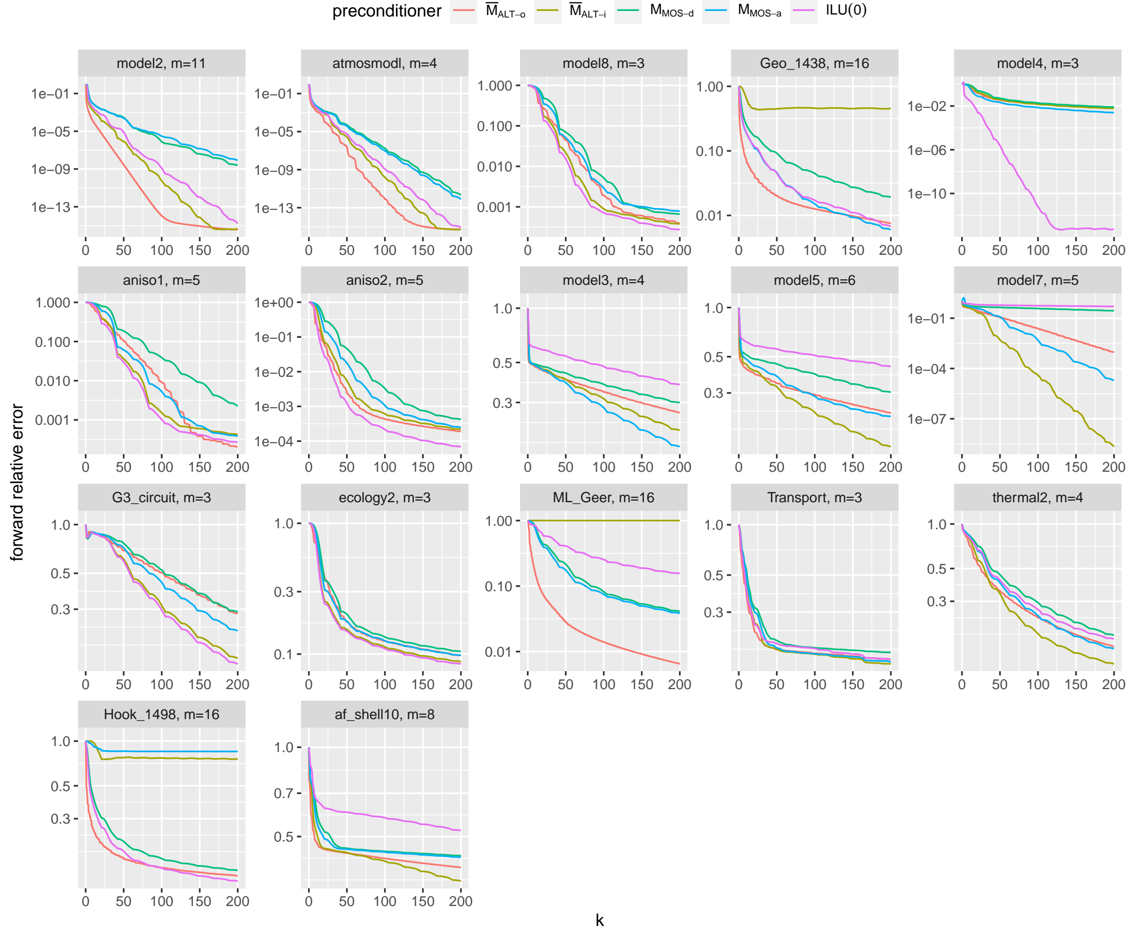


Figure 40.: Full double-precision GMRES with forward relative error on logarithmic y-scale against number of iterations for operator split preconditioners (with a weight coverage of at least 99%) in comparison to an exactly solved ILU(0). Exceptionally, the scheme $\overline{M}_{\text{ALT-o}}$ (Eq. 35) performs m iterations for each k.

5.3.5. Comparison with exactly solved ILU(0)

For the performance analysis in Section 5.3.3, we have compared the operator split preconditioners with the ILU(0)-ISAI(m) preconditioner within a mixed precision iterative refinement. In this section, we disregard some performance considerations for a clean comparison of the convergence behavior.

We now invert the ILU(0) exactly and execute everything in full double-precision, running GMRES with a restart of 20 as an outer solver. Moreover, for the operator split preconditioners, we choose m such that 99% of the matrix weight is included (Eq. 53). In this way, the operator split preconditioners use approximately the same number of coefficients as ILU(0). For $M_{\text{MOS-d}}$ this means that the error term R_1 is close to zero in Eq. 66.

Figure 40 shows the convergence results for the test matrices of Table 17 with the m value in each header. For 8 out of 17 matrices, the $M_{\text{MOS-a}}$ and $\overline{M}_{\text{ALT-i}}$ preconditioners show superior convergence to ILU(0). Moreover, the above procedure to determine m is actually too simplistic: $\overline{M}_{\text{ALT-i}}$ convergence better on matrices GEO_1438, ML_GEER and HOOK_1498 with a smaller m (see Figures 37 and 38).

5.4. Conclusions

We have introduced four approaches for operator splitting preconditioners and a new diagonal preconditioner for general sparse matrices and evaluated the special case with tridiagonal splittings in detail with GMRES as the outer solver. This special case already shows superior convergence against ILU. On the difficult problems the better convergence also translates to faster execution. However, we must develop better understanding of theory and practice for these new schemes. With specifically designed prolongation and restriction operators, they might also be useful as smoothers in algebraic multigrid solvers. Beyond tridiagonal splittings, the general algebraic framework opens up many opportunities and this will be explored in the future.

5.5. Other Contributors

Dr. Wayne Mitchell created the problem statements from Hypre and MFEM in Table 17.

6. Conclusion

In this thesis, a multi-functional (block) tridiagonal CUDA GPU library, a generalized sparse matrix-vector product (SpMV) implementation, a weighted linear forest extraction & $[0, n]$-factor computation, and multiplicative operator split preconditioners have been presented. Each contribution improves state-of-the-art solutions: The tridiagonal library outperforms cuSPARSE's scalar tridiagonal solver by factor five, and also supports block tridiagonal systems. The SpMV implementation achieves the same performance as cuSPARSE while providing a much larger generality, such that it can be used for more complex problem statements like $[0, n]$-factor computations. An essential contribution of the linear forest extraction is the parallel bidirectional scan, which only requires a forward iterator type (support for `operator++` and `operator--`), and thus can operate on double-linked lists. The algebraic operator splitting is a generalization and improvement of ILU methods for some sparse matrices in case of tridiagonal splittings.

This thesis evaluated stronger GPU thread data privatization methods, which do not expose parallelism as fine-grained as possible but rather use the work per GPU thread as a tuning parameter. This data privatization was implemented in the tridiagonal and SpMV library such that the data is loaded coalesced with all threads of the CUDA block into shared memory, and processed subsequently only by one warp per CUDA block. With an appropriate choice of the work per GPU thread, the benchmarks showed that stronger data privatization achieves the same performance as fine-grained parallelization techniques, and thus should be considered as viable method for GPU algorithm design as it also enables the partial usage of sequential algorithms in GPU kernels, e.g., the (scaled) partial pivoting in the tridiagonal library.

In sparse linear algebra, most problems are memory limited, which makes efficiently used shared memory and data locality essential for the development of highest performance kernels. Therefore, the kernels of the tridiagonal library and the SpMV implementation were carefully tuned with respect to these aspects, e.g., the tridiagonal kernels diagonalize the system with on-chip data only and the SpMV implementation supports several user types for each memory location (global memory, shared memory, registers) to avoid unecessary data movement.

Moreover, it has been shown that the efficient and general solution of building-blocks like the SpMV implementation and the tridiagonal solver, open up more advanced use-cases like linear forest extraction and algebraic tridiagonal preconditioners, respectively.

There are many directions in which the work of this thesis might be continued. In general, multi-GPU support and a larger set of valid input parameters would be useful, e.g., support for banded systems with more than nine bands in the tridiagonal library, or $[0, n]$-factor computations with $n > 4$. However, this is not only a little change in the implementation but requires a new algorithmic formulation. Regarding the presented algebraic operator split preconditioners, more approximative construction & factorization schemes, and a better understanding of the preconditioner limitations and strengths should be investigated in the future.

References

Most parts of this thesis have already been published, which is indicated by a note at the beginning of a chapter. Chapter 2 is based on [72] and [75], Chapter 3 is new, Chapter 4 is based on [73], and Chapter 5 is based on [74].

A. Bibliography

[1] Tolu Alabi, Russel Steinbach, Jeffrey D. Blanchard, and Bradley Gordon, *Fast k-Selection Algorithms for Graphics Processing Units*, ACM Journal of Experimental Algorithmics **17** (2012), no. 4, 4.1–4.29.

[2] Pablo Alfaro, Pablo Igounet, and Pablo Ezzatti, *A study on the implementation of tridiagonal systems solvers using a GPU*, Proceedings - International Conference of the Chilean Computer Science Society, SCCC, IEEE, nov 2012, pp. 219–227.

[3] Noga Alon, V. J. Teague, and N. C. Wormald, *Linear Arboricity and Linear k-Arboricity of Regular Graphs*, Graphs and Combinatorics **17** (2001), no. 1, 11–16.

[4] P. Amodio and L. Brugnano, *Parallel factorizations and parallel solvers for tridiagonal linear systems*, Linear Algebra and Its Applications **172** (1992), no. C, 347–364.

[5] E Anderson, Z Bai, C Bischof, S Blackford, J Demmel, J Dongarra, J Du Croz, A Greenbaum, S Hammarling, A McKenney, and D Sorensen, *LAPACK Users' Guide*, third ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999.

[6] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, Will Pazner, Mark Stowell, Vladimir Tomov, Ido Akkerman, Johann Dahm, David Medina, and Stefano Zampini, *MFEM: A modular finite element methods library*, Computers and Mathematics with Applications **81** (2021), 42–74.

[7] Hartwig Anzt, Thomas K. Huckle, Jürgen Bräckle, and Jack Dongarra, *Incomplete Sparse Approximate Inverses for Parallel Preconditioning*, Parallel Computing **71** (2018), 1–22.

[8] Hartwig Anzt, William Sawyer, Stanimire Tomov, Piotr Luszczek, Ichitaro Yamazaki, and Jack Dongarra, *Optimizing Krylov subspace solvers on graphics processing units*, Proceedings - IEEE 28th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2014 (Phoenix, AZ), IEEE, IEEE, 2014, pp. 941–949.

[9] Arash Ashari, Naser Sedaghati, John Eisenlohr, Srinivasan Parthasarathy, and P. Sadayappan, *Fast Sparse Matrix-Vector Multiplication on GPUs for Graph Applications*, International Conference for High Performance Computing, Networking, Storage and Analysis, SC **2015-Janua** (2014), no. January, 781–792.

[10] Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang, *Multigrid smoothers for ultraparallel computing*, SIAM Journal on Scientific Computing **33** (2011), no. 5, 2864–2887.

[11] Nathan Bell and Michael Garland, *Efficient Sparse Matrix-Vector Multiplication on CUDA*, Memory (2008), 1–32.

[12] ———, *Implementing sparse matrix-vector multiplication on throughput-oriented processors*, Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09 (New York, New York, USA), vol. 30, ACM Press, jul 2009, p. 1.

[13] Sébastien Bougleux and Luc Brun, *Linear Sum Assignment with Edition*, (2016), no. 1, 1–27.

[14] Rainer E. Burkard and Ulrich Derigs, *The Linear Sum Assignment Problem*, (1980), 1–15.

[15] Zhiqiang Cao and Hong Xue, *United States Patent : 6599524 United States Patent : 6599524*, 2014, pp. 2–7.

[16] ———, *United States Patent : 6599524 United States Patent : 6599524*, 2014, pp. 2–7.

[17] ———, *United States Patent : 6599524 United States Patent : 6599524*, 2014, pp. 2–7.

[18] Li-Wen Chang, *Scalable parallel tridiagonal algorithms with diagonal pivoting and their optimization for many-core architectures*, Ph.D. thesis, 2014.

[19] Li Wen Chang, John A. Stratton, Hee Seok Kim, and Wen Mei W. Hwu, *A scalable, numerically stable, high-performance tridiagonal solver using GPUs*, International Conference for High Performance Computing, Networking, Storage and Analysis, SC (2012).

[20] Xiaojun Chen and Weijun Zhou, *Smoothing nonlinear conjugate gradient method for image restoration using nonsmooth nonconvex minimization*, SIAM Journal on Imaging Sciences **3** (2010), no. 4, 765–790.

[21] Yen-shen Chen and Yu-chin Hsu, *Based on Bipartite Weighted Matching f*, (1990), no. 1.

[22] Zhi Zhong Chen, *Fundamental Study Parallel constructions of maximal path sets and applications to short superstrings*, Theoretical Computer Science **161** (1996), no. 1-2, 1–21.

[23] Edmond Chow and Aftab Patel, *Fine-grained parallel incomplete LU factorization*, SIAM Journal on Scientific Computing **37** (2015), no. 2, C169–C193.

[24] Jonathan Cohen, *Efficient Graph Matching and Coloring on the GPU*, Gtc, 2012, p. 58.

[25] Ketan Date and Rakesh Nagi, *GPU-accelerated Hungarian algorithms for the Linear Assignment Problem*, Parallel Computing **57** (2016), 52–72.

[26] Andrew Davidson, Yao Zhang, and John D. Owens, *An auto-tuned method for solving large tridiagonal systems on the GPU*, Proceedings - 25th IEEE International Parallel and Distributed Processing Symposium, IPDPS 2011 (2011), 956–965.

[27] Timothy A. Davis and Yifan Hu, *The University of Florida Sparse Matrix Collection*, ACM Transactions on Mathematical Software **38** (2011), no. 1.

[28] Gianna M. Del Corso, *Estimating an eigenvector by the power method with a random start*, SIAM Journal on Matrix Analysis and Applications **18** (1997), no. 4, 913–937.

[29] Adrián P. Diéguez, Margarita Amor, and Ramón Doallo, *Parallel prefix operations on GPU: tridiagonal system solvers and scan operators*, Journal of Supercomputing **75** (2019), no. 3, 1510–1523.

[30] ______, *Tree partitioning reduction: A new parallel partition method for solving tridiagonal systems*, ACM Transactions on Mathematical Software **45** (2019), no. 3.

[31] Adrian Perez Dieguez, Margarita Amor, and Ramon Doallo, *New Tridiagonal Systems Solvers on GPU Architectures*, Proceedings - 22nd IEEE International Conference on High Performance Computing, HiPC 2015 (2016), 85–93.

[32] Adrián Pérez Diéguez, Margarita Amor, Jacobo Lobeiras, and Ramón Doallo, *Solving large problem sizes of index-digit algorithms on GPU: FFT and tridiagonal system solvers*, IEEE Transactions on Computers **67** (2018), no. 1, 86–101.

[33] J. Douglas, R. B. Kellogg, and R. S. Varga, *Alternating Direction Iteration Methods for n Space Variables*, Mathematics of Computation **17** (1963), no. 83, 279.

[34] Jim Douglas, *Alternating direction methods for three space variables*, Numerische Mathematik **4** (1962), no. 1, 41–63.

[35] J Douglas Jr and T Dupont, *Alternating direction Galerkin methods on rectangles, Numerical Solution of Partial Differential Equations-II (B. Hubbard, ed.)*, 1971.

[36] I. S. Duff, *Algorithm 575: Permutations for a Zero-Free Diagonal [F1]*, ACM Transactions on Mathematical Software (TOMS) **7** (1981), no. 3, 387–390.

[37] ______, *On Algorithms for Obtaining a Maximum Transversal*, ACM Transactions on Mathematical Software (TOMS) **7** (1981), no. 3, 315–330.

[38] I. S. Duff and J. Koster, *On algorithms for permuting large entries to the diagonal of a sparse matrix*, SIAM Journal on Matrix Analysis and Applications **22** (2001), no. 4, 973–996.

[39] Iain S. Duff, Kamer Kaya, and Bora Uçar, *Design, implementation, and analysis of maximum transversal algorithms*, ACM Transactions on Mathematical Software **38** (2011), no. 2.

[40] Iain S. Duff and Jacko Koster, *The design and use of algorithms for permuting large entries to the diagonal of sparse matrices*, SIAM Journal on Matrix Analysis and Applications **20** (1999), no. 4, 889–901.

[41] J.W. Eastwood, *Parallel computers: Architecture, programming and algorithms*, vol. 27, Bristol, 1982.

[42] Jennifer B. Erway, Roummel F. Marcia, and Joseph A. Tyson, *Generalized diagonal pivoting methods for tridiagonal systems without interchanges*, IAENG International Journal of Applied Mathematics **40** (2010), no. 4, 1–7.

[43] Ömer Eğecioğlu, Cetin K. Koc, and Alan J. Laub, *A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors*, Journal of Computational and Applied Mathematics **27** (1989), no. 1-2, 95–108.

[44] Bas O. Fagginger Auer and Rob H. Bisseling, *A GPU Algorithm for Greedy Graph Matching*, (2012), 108–119.

[45] Robert D. Falgout and Ulrike Meier Yang, *hypre: A library of high performance preconditioners*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **2331 LNCS** (2002), no. PART 3, 632–641.

[46] Salvatore Filippone, Valeria Cardellini, Davide Barbieri, and Alessandro Fanfarillo, *Sparse matrix-vector multiplication on GPGPUs*, ACM Transactions on Mathematical Software **43** (2017), no. 4.

[47] Goran Flegar and Enrique S. Quintana-Ortí, *Balanced CSR sparse matrix-vector product on graphics processors*, Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) **10417 LNCS** (2017), 697–709.

[48] S Foulon and K.J. in 't Hout, *Adi Finite Difference Schemes for Option Pricing*, International Journal of Numerical Analysis and Modeling **7** (2010), no. 2, 303–320.

[49] ABHIJIT GHOSH and CHITTARANJAN MISHRA, *A parallel cyclic reduction algorithm for pentadiagonal systems with application to a convection-dominated heston pde*, SIAM Journal on Scientific Computing **43** (2021), no. 2, C177–C202.

[50] Mike Giles, Endre Laszlo, Istvan Reguly, Jeremy Appleyard, and Julien Demouth, *GPU Implementation of Finite Difference Solvers*, Proceedings of WHPCF 2014: 7th Workshop on High Performance Computational Finance - Held in conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis (2014), 1–8.

[51] Andrew Gloster, Lennon Ó Náraigh, and Khang Ee Pang, *cuPentBatch—A batched pentadiagonal solver for NVIDIA GPUs*, Computer Physics Communications **241** (2019), 113–121.

[52] Wotao Yin Glowinski, Roland and Stanley J Osher, *Splitting Methods in Communication , Imaging , Science ,*, Springer, 2016.

[53] Dominik Göddeke and Robert Strzodka, *Cyclic reduction tridiagonal solvers on GPUs applied to mixed-precision multigrid*, IEEE Transactions on Parallel and Distributed Systems **23** (2011), no. 1, 22–32.

[54] Dominik Göddeke, Robert Strzodka, and Stefan Turek, *Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations*, International Journal of Parallel, Emergent and Distributed Systems **22** (2007), no. 4, 221–256.

[55] S. González-Pinto and D. Hernández-Abreu, *Splitting-methods based on Approximate Matrix Factorization and Radau-IIA formulas for the time integration of advection diffusion reaction PDEs*, Applied Numerical Mathematics **104** (2016), 166–181.

[56] Joseph L. Greathouse and Mayank Daga, *Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format*, International Conference for High Performance Computing, Networking, Storage and Analysis, SC **2015-Janua** (2014), no. January, 769–780.

[57] Gaël Guennebaud, Benoît Jacob, and Others, *Eigen v3*, http://eigen.tuxfamily.org, 2012.

[58] Stefan Guthe and Daniel Thuerck, *Algorithm 1015 A Fast Scalable Solver for the Dense Linear (Sum) Assignment Problem*, ACM Transactions on Mathematical Software **47** (2021), no. 2.

[59] Michael Hagemann and Olaf Schenk, *Weighted matchings for preconditioning symmetric indefinite linear systems*, SIAM Journal on Scientific Computing **28** (2006), no. 2, 403–420.

[60] George R. Halliwell, *Evaluation of vertical coordinate and vertical mixing algorithms in the HYbrid-Coordinate Ocean Model (HYCOM)*, Ocean Modelling **7** (2004), no. 3-4, 285–322.

[61] Frank Harary, *COVERING AND PACKING IN GRAPHS, I.*, Annals of the New York Academy of Sciences **175** (1970), no. 1, 198–205.

[62] Jared Hoberock and Nathan Bell, *Thrust: A parallel template library*, 2010, p. 43.

[63] R. W. Hockney, *A Fast Direct Solution of Poisson's Equation Using Fourier Analysis*, Journal of the ACM (JACM) **12** (1965), no. 1, 95–113.

[64] Jonathan Hogg and Jennifer Scott, *On the use of suboptimal matchings for scaling and ordering sparse symmetric matrices*, Numerical Linear Algebra with Applications **22** (2015), no. 4, 648–663.

[65] Willem Hundsdorfer and Jan G Verwer, *Numerical solution of time-dependent advection-diffusion-reaction equations*, vol. 33, Springer Science & Business Media, 2013.

[66] Rabia Kamra and S. Chandra Sekhara Rao, *A stable parallel algorithm for block tridiagonal toeplitz–block–toeplitz linear systems*, Mathematics and Computers in Simulation **190** (2021), 1415–1440.

[67] Michael Kass, Aaron Lefohn, and John Owens, *Interactive Depth of Field Using Simulated Diffusion on a GPU*, Computing (2006), no. January 2006, 1–8.

[68] Michael Kass and Gavin Miller, *Rapid, stable fluid dynamics for computer graphics*, **24** (1990), no. 4, 49–57.

[69] Hee Seok Kim, Shengzhao Wu, Li Wen Chang, and Wen Mei W. Hwu, *A scalable tridiagonal solver for GPUs*, Proceedings of the International Conference on Parallel Processing (2011), 444–453.

[70] Ki Ha Kim, Ji Hoon Kang, Xiaomin Pan, and Jung Il Choi, *PaScaL_TDMA: A library of parallel and scalable solvers for massive tridiagonal systems*, Computer Physics Communications **260** (2021), 107722.

[71] Christoph Klein and Robert Strzodka, *tridigpu, https://mp-force.ziti.uni-heidelberg.de/asc/code/tridigpu.*

[72] ———, *Tridiagonal GPU Solver with Scaled Partial Pivoting at Maximum Bandwidth*, ACM International Conference Proceeding Series (New York, NY, USA), ACM, aug 2021, pp. 1–10.

[73] ______, *Highly Parallel Linear Forest Extraction from a Weighted Graph on GPUs*, ACM, ICPP, 2022.

[74] ______, *Preconditioning Sparse Matrices with Alternating and Multiplicative Operator Splittings [ACCEPTED AND IN PRODUCTION]*, SISC (2023).

[75] ______, *tridigpu: A GPU library for block tridiagonal and banded linear equation systems [ACCEPTED AND IN PRODUCTION]*, TOPC (2023).

[76] Ki Bok Kong, Yong Sun Shin, Seong Ook Park, and Jong Sung Kim, *The simplest V-cycle fast adaptive composite grid ADI-FDTD method for two-dimensional electromagnetic simulations*, IEEE Antennas and Wireless Propagation Letters **7** (2008), 451–455.

[77] Endre László, Mike Giles, and Jeremy Appleyard, *Manycore algorithms for batch scalar and block tridiagonal solvers*, ACM Transactions on Mathematical Software **42** (2016), no. 4.

[78] Jingmei Li, Zhigao Zheng, Qiao Tian, Guoyin Zhang, Fangyuan Zheng, and Yuanyuan Pan, *Research on tridiagonal matrix solver design based on a combination of processors*, Computers and Electrical Engineering **62** (2017), 1–16.

[79] Yuan-Chuan Li and Cheh-Chih Yeh, *Some Equivalent Forms of Bernoulli's Inequality: A Survey*, Applied Mathematics **04** (2013), no. 07, 1070–1093.

[80] Yongchao Liu and Bertil Schmidt, *LightSpMV: Faster CUDA-Compatible Sparse Matrix-Vector Multiplication Using Compressed Sparse Rows*, Journal of Signal Processing Systems **90** (2018), no. 1, 69–86.

[81] J. Lobeiras, M. Amor, and R. Doallo, *BPLG: A Tuned Butterfly Processing Library for GPU Architectures*, International Journal of Parallel Programming **43** (2015), no. 6, 1078–1102.

[82] Timothy G. Mattson, Carl Yang, Scott McMillan, Aydin Buluc, and Jose E. Moreira, *GraphBLAS C API: Ideas for future versions of the specification*, 2017 IEEE High Performance Extreme Computing Conference, HPEC 2017 (2017), 1–6.

[83] D. J. Mavriplis, *Multigrid Strategies for Viscous Flow Solvers on Anisotropic Unstructured Meshes*, Journal of Computational Physics **145** (1998), no. 1, 141–165.

[84] Duane Merrill, *CUB*, 2021.

[85] Duane Merrill and Michael Garland, *Merge-based sparse matrix-vector multiplication (SpMV) using the CSR storage format*, ACM SIGPLAN Notices **51** (2016), no. 8, 1–2.

[86] ______, *Single-pass Parallel Prefix Scan with Decoupled Look-back*, NVIDIA Technical Report NVR-2016-002 (2016), 1–9.

[87] Gordon E. Moore, *Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.*, IEEE Solid-State Circuits Society Newsletter **11** (2006), no. 3, 33–35.

[88] Md Naim, Fredrik Manne, Mahantesh Halappanavar, Antonino Tumeo, and Johannes Langguth, *Optimizing Approximate Weighted Matching on NVIDIA Kepler K40*, Proceedings - 22nd IEEE International Conference on High Performance Computing, HiPC 2015 (2016), 105–114.

[89] M. Naumov, M. Arsaev, P. Castonguay, J. Cohen, J. Demouth, J. Eaton, S. Layton, N. Markovskiy, I. Reguly, N. Sakharnykh, V. Sellappan, and R. Strzodka, *AMGX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods*, SIAM Journal on Scientific Computing **37** (2015), no. 5, S602–S626.

[90] C NVIDIA, *CUSPARSE library*, NVIDIA Corporation, Santa Clara, California, no. July, 2011.

[91] D. W. Peaceman and H. H. Rachford, Jr., *The Numerical Solution of Parabolic and Elliptic Differential Equations*, Journal of the Society for Industrial and Applied Mathematics **3** (1955), no. 1, 28–41.

[92] Carl Pearcy, *On convergence of alternating direction procedures*, Numerische Mathematik **4** (1962), no. 1, 172–176.

[93] ———, *On convergence of alternating direction procedures*, Numerische Mathematik **4** (1962), no. 1, 172–176.

[94] John W. Pearson and Jennifer Pestana, *Preconditioners for Krylov subspace methods: An overview*, GAMM Mitteilungen **43** (2020), no. 4, 1–35.

[95] Hannu Peltola, Hans Soderlund, Jorma Tarhio, and Esko Ukkonen, *Algorithms for Some String Matching Problems Arising in Molecular Genetics.*, IFIP Congress Series, vol. 9, 1983, pp. 59–64.

[96] Adrian Perez DIeguez, Margarita Amor Lopez, and Ramon Doallo Biempica, *Solving Multiple Tridiagonal Systems on a Multi-GPU Platform*, Proceedings - 26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2018 (2018), 759–763.

[97] Bobby Philip and Timothy P. Chartier, *Adaptive algebraic smoothers*, Journal of Computational and Applied Mathematics **236** (2012), no. 9, 2277–2297.

[98] Michael D. Plummer, *Graph factors and factorization: 1985-2003: A survey*, Discrete Mathematics **307** (2007), no. 7-8, 791–821.

[99] Eric Polizzi and Ahmed Sameh, *SPIKE: A parallel environment for solving banded linear systems*, Computers and Fluids **36** (2007), no. 1, 113–120.

[100] Eric Polizzi and Ahmed H. Sameh, *A parallel hybrid banded system solver: The SPIKE algorithm*, Parallel Computing **32** (2006), no. 2, 177–194.

[101] M. Prieto, R. Santiago, D. Espadas, I. M. Llorente, and F. Tirado, *Parallel Multigrid for Anisotropic Elliptic Equations*, Journal of Parallel and Distributed Computing **61** (2001), no. 1, 96–114.

[102] Thomas H. Pulliam, *Implicit solution methods in computational fluid dynamics*, Applied Numerical Mathematics **2** (1986), no. 6, 441–474.

[103] Roberto Roverso, Amgad Naiem, Mohammed El-Beltagy, Sameh El-Ansary, and Seif Haridi, *A GPU-enabled solver for time-constrained linear sum assignment problems*, INFOS2010 - 2010 7th International Conference on Informatics and Systems (2010).

[104] Yousef Saad, *Iterative Methods for Sparse Linear Systems*, Iterative Methods for Sparse Linear Systems (2003).

[105] Takayoshi Shoudai and Satoru Miyano, *Using maximal independent sets to solve problems in parallel*, Theoretical Computer Science **148** (1995), no. 1, 57–65.

[106] Brandon Skerritt, *Graph Theory Graph theory*, 5th ed., Springer-Verlag, Heidelberg, 2018.

[107] Harold S. Stone, *An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations*, Journal of the ACM (JACM) **20** (1973), no. 1, 27–38.

[108] Eng Leong Tan, *Fundamental schemes for efficient unconditionally stable implicit finite-difference time-domain methods*, IEEE Transactions on Antennas and Propagation **56** (2008), no. 1, 170–177.

[109] L Thomas, *Elliptic problems in linear differential equations over a network*, Watson Sci. Comput. Lab Report, Columbia University, New York **1** (1949), 42.

[110] Ryuhei Uehara and Zhi Zhong Chen, *Parallel algorithms for maximal linear forests*, 1997, pp. 627–634.

[111] P. J. Van Der Houwen and B. P. Sommeijer, *Approximate factorization for time-dependent partial differential equations*, Journal of Computational and Applied Mathematics **128** (2001), no. 1-2, 447–466.

[112] Henk A. van der Vorst, *Large tridiagonal and block tridiagonal linear systems on vector and parallel computers*, Parallel Computing **5** (1987), no. 1-2, 45–54.

[113] I. E. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, and A. H. Sameh, *A direct tridiagonal solver based on Givens rotations for GPU architectures*, Parallel Computing **49** (2015), 101–116.

[114] Eugene Wachspress, *The ADI model problem*, vol. 9781461451, Springer, 2013.

[115] H. H. Wang, *A Parallel Method for Tridiagonal Equations*, ACM Transactions on Mathematical Software (TOMS) **7** (1981), no. 2, 170–183.

[116] Xiaojing Wang and Z. G. Mou, *A divide-and-conquer method of solving tridiagonal systems on hypercube massively parallel computers*, Proceedings of the 3rd IEEE Symposium on Parallel and Distributed Processing 1991 (1991), 810–817.

[117] Bebo White, *Amy Langville and Carl Meyer, Google's Page Rank and Beyond: The Science of Search Engine Rankings*, Information Retrieval **11** (2008), no. 5, 471–472.

[118] Technical Whitepaper, *Semantic Versioning Semantic Versioning*, 2010, pp. 1–10.

[119] Calvin H. Wilcox, *Perturbation Theory for Linear Operators (Tosio Kato)*, vol. 12, Springer Science & Business Media, 1970.

[120] Guoliang Xing, Chenyang Lu, Ying Zhang, Qingfeng Huang, and Robert Pless, *Minimum power configuration for wireless communication in sensor networks*, ACM Transactions on Sensor Networks **3** (2007), no. 2, 1–33.

[121] Hong Zhang, Adrian Sandu, and Paul Tranquilli, *Application of approximate matrix factorization to high order linearly implicit Runge-Kutta methods*, Journal of Computational and Applied Mathematics **286** (2015), 196–210.

[122] Yao Zhang, Jonathan Cohen, and John D. Owens, *Fast tridiagonal solvers on the GPU*, ACM SIGPLAN Notices **45** (2010), no. 5, 127–136.

[123] Zhengyong Zhu, Rui Shi, Chung Kuan Cheng, and Ernest S. Kuh, *An unconditional stable general operator splitting method for transistor level transient analysis*, Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, vol. 2006, 2006, pp. 428–433.

A. Appendix

A.1. Tridiagonal Library

tridigpu [71] is published under the the 3-Clause BSD License and is intended to be backward compatible at the source level, and follows the semantic versioning guidelines introduced by Tom Preston-Werner [118].

A.1.1. Naming Conventions

We follow the general BLAS naming conventions and provide a C-API

tridigpu<t><format><operation>,

where <t> can be S, D, C, Z, which represent the types float, double, cuComplex, and cuDoubleComplex, respectively. As a placeholder for the corresponding types we use <T>. We use [c] to indicate an optional letter c in the function name, which refers to the version of the function for cyclic systems.

A.1.2. Data Format and Layouts

Dense matrices

All dense matrices are saved in column-major order. Therefore, the multiple right-hand sides X are saved consecutively in memory. E.g. the right-hand sides of Equation 3 are saved as

```
D = [1, 2, 3, 4, 5, 6, 7, 8, ..., 17, 18].
```

Tridiagonal matrices

The tridiagonal system is saved in a banded format, which are three separate buffers (a_i, b_i, c_i) each of length $\hat{N}n^2$. Thus, matrix A of Equation 2 is saved as

```
a = [ 1,  2,  3,  4,  5,  6]
b = [ 7,  8,  9, 10, 11, 12]
c = [13, 14, 15, 16, 17, 18],
```

whereas a[0] = 0 and c[5] = 0 for the tridiagonal system of Equation 3.

Block tridiagonal matrices

For block tridiagonal systems, each block is saved as a dense matrix in column-major order, and the tridiagonal bands of Equation 4 are saved as

```
a = [ 0,  0,  0,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12]
b = [13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]
c = [29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,  0,  0,  0,  0].
```

Banded matrices

We save banded matrices from the the lowest to the highest band. . Thus, the banded matrix from Equation 5 is saved as

```
kl = ku = 2
AB = [ *,  *,  1,  4,  5,  8,  9, 12,
        *, 14,  3, 18,  7, 22, 11, 26,
       13, 16, 17, 20, 21, 24, 25, 28,
       15, 30, 19, 34, 23, 38, 27,  *,
       29, 32, 33, 36, 37, 40,  *,  *].
```

Analogously to the scalar case in Section A.1.2, the placeholders * contain coefficients in case of a cyclic system and are unused otherwise. The buffer has a length of $N(k_l + k_u + 1)$, where k_l, k_u is the number of bands below and above the diagonal band, respectively. The diagonal is always included in this format, and the distances of the bands to the diagonal are fixed, e.g., for the lower bands: $-k_l, -(k_l - 1), \ldots, -1$.

Sparse matrices

Sparse matrices are saved in a compressed sparse column format (CSC) with zero-based indexing (see CSC format in cuSPARSE [90]). The CSC matrix $\mathcal{D}$ of Equation 6 is saved as

```
values =      [1,  5, 10, 11,  2, 12,  3,  8, 13,  6,  9,  4,  7]
row_indices = [0,  1,  5,  6,  0,  6,  0,  3,  6,  1,  3,  0,  1]
col_ptrs = [0, 4, 6, 9, 11, 12, 13].
```

A.1.3. Resource Consumption

RPTS is a hierarchical algorithm, which must save intermediate results during the processing of higher stages. The amount of required temporary memory depends on various parameters like the data type, the partition size, the number of stages in the hierarchy, the size of the blocks, and the number of right-hand sides. When the original tridiagonal equation system size is calculated with $N(n_r + 3)$, a scalar tridiagonal solve in single-precision with `tridigpuSgtsv` requires approximately between 5% and 11% as temporary memory, whereas `cusparseSgtsv2` [90] requires between 100% and 110% of the tridiagonal system byte size. Moreover, `tridigpu` does not require any global initialization or finalization functions.

A.1.4. C API

In this section the function declarations of the C-API of `tridigpu` are presented. Most functions take a parameter to control the pivoting scheme, which is an `enum` of type `tridigpu_pivoting_t` with the instances `TRIDIGPU_NO_PIVOTING`, `TRIDIGPU_PARTIAL_PIVOTING`, `TRIDIGPU_SCALED_PARTIAL_PIVOTING`, and `TRIDIGPU_DEFAULT_PIVOTING`. Many functions require a (temporary) buffer of GPU memory for execution, the size of which is returned by the corresponding tridigpu<t><format><operation>_bufferSizeExt function.

Solve (cyclic) scalar tridiagonal systems with multiple right-hand sides

This function solves Problem Class Cyclic and Problem Class Scalar, which are (cyclic) general tridiagonal systems.

```
tridigpu_status_t
tridigpu<t>[c]gtsv_bufferSizeExt(tridigpu_pivoting_t p, int N, int n_r, size_t*
    buffer_size_in_bytes)

tridigpu_status_t
tridigpu<t>[c]gtsv(cudaStream_t stream, tridigpu_pivoting_t p,
                   int          N, int        n_r,
                   const <T>*   a, const <T>* b, const <T>* c,
                   <T>*         X, const <T>* D,
                   void*        p_buffer)
```

The function does not allocate any extra storage, enqueues the calculation into the CUDA stream, and is non-blocking, i.e., it may return control back to the host before the result is ready. For the solution of cyclic systems, `cgtsv` is provided.

Input	Description
`p`	recommended value: `TRIDIGPU_DEFAULT_PIVOTING` (see above)
`N`	size of the tridiagonal system
`n_r`	number of right-hand sides
`a`	lower band of tridiagonal matrix (length `N`)
`b`	diagonal of tridiagonal matrix (length `N`)
`c`	upper diagonal of tridiagonal matrix (length `N`)
`D`	dense right-hand side matrix and buffer of length `N * n_r`
`p_buffer`	temporarily required memory buffer on the GPU

Output	Description
`X`	dense solution matrix and buffer of length `N * n_r`. Buffer `D` may be passed also as `X`, then the solution replaces the right-hand side data after kernel execution.

Example:

```
size_t buffer_size_in_bytes = 0;
if (tridigpuSgtsv_bufferSizeExt(p, N, n_r, &buffer_size_in_bytes)) {
  printf("determining temporary buffer size failed\n");
}
void* p_buffer;
if (cudaMalloc(&p_buffer, buffer_size_in_bytes)) {
  printf("allocating temporary buffer failed\n");
}
if (tridigpuSgtsv(stream, p, N, n_r, a, b, c, X, D, p_buffer)) {
  printf("solving tridiagonal system failed\n");
}
```

Solve block tridiagonal systems with multiple right-hand sides

The functions in this Section solve Problem Class Block.

```
tridigpu_status_t
tridigpu<t>[c]bgtsv_bufferSizeExt(tridigpu_pivoting_t p, int N, int n_r, int n,
    size_t* buffer_size_in_bytes)

tridigpu_status_t
tridigpu<t>[c]bgtsv(cudaStream_t stream, tridigpu_pivoting_t p,
                    int          N, int        n_r, int      n,
                    const <T>*   a, const <T>* b, const <T>* c,
                    <T>*         X, const <T>* D,
                    void*        p_buffer)
```

The function does not allocate any extra storage, enqueues the calculation into the CUDA stream, and is non-blocking, i.e., it may return control back to the host before the result is ready. The function is available for block (`bgtsv`) and cyclic block general tridiagonal systems (`cbgtsv`).

Input	Description
`p`	recommended value: `TRIDIGPU_DEFAULT_PIVOTING` (see above)
`N`	size of the block tridiagonal system
`n_r`	number of right-hand sides
`n`	size of one block; allowed values are 1, 2, 3, 4
`a`	lower band of tridiagonal matrix (length `N * n * n`)
`b`	diagonal of tridiagonal matrix (length `N * n * n`)
`c`	upper diagonal of tridiagonal matrix (length `N * n * n`)
`D`	dense right-hand side matrix and buffer of length `N * n_r * n`
`p_buffer`	temporarily required memory buffer on the GPU

Output	Description
`X`	dense solution matrix and buffer of length `N * n_r * n`. Buffer `D` may be passed also as `X`, then the solution replaces the right-hand side data after kernel execution.

Example:

```
size_t buffer_size_in_bytes = 0;
if (tridigpuSbgtsv_bufferSizeExt(p, N, n_r, n, &buffer_size_in_bytes)) {
  printf("determining temporary buffer size failed\n");
}
void* p_buffer;
if (cudaMalloc(&p_buffer, buffer_size_in_bytes)) {
  printf("allocating temporary buffer failed\n");
}
if (tridigpuSbgtsv(stream, p, N, n_r, n, a, b, c, X, D, p_buffer)) {
  printf("solving block tridiagonal system failed\n");
}
```

Function bgtsv factorizes the matrix with every function call, whereas bgtf calculates the factorization explicitly once, and bgtfs[1] uses it to compute the solution from a given right-hand side. The explicit factorization is more efficient if the kernels of tridigpu are not memory bound any more.

```
tridigpu_status_t
tridigpu<t>bgtf_factorizationBufferSizeExt(tridigpu_pivoting_t p, int N, int n,
    size_t* factorization_buffer_size_in_bytes)

tridigpu_status_t
tridigpu<t>bgtf(cudaStream_t stream, tridigpu_pivoting_t    p,
                int          N, int         n,
                <T>*         a, <T>*        b, <T>*         c,
                void*        p_factorization_buffer)
```

The function does not allocate any extra storage, enqueues the calculation into the CUDA stream, and is non-blocking, i.e., it may return control back to the host before the result is ready.

Input	Description
p	recommended value: TRIDIGPU_DEFAULT_PIVOTING (see above)
N	size of the block tridiagonal system
n	size of one block; allowed values are 1, 2, 3, 4
a	lower band of tridiagonal matrix (length N * n * n)
b	diagonal of tridiagonal matrix (length N * n * n)
c	upper diagonal of tridiagonal matrix (length N * n * n)

Output	Description
a	lower band of factorized tridiagonal matrix (length N * n * n)
b	diagonal of factorized tridiagonal matrix (length N * n * n)
c	upper diagonal of factorized tridiagonal matrix (length N * n * n)
p_factorization_buffer	buffer to save additional data required to factorize the matrix

The tridiagonal system is solved for a given factorization and a corresponding right-hand side with bgtfs.

```
tridigpu_status_t
tridigpu<t>bgtfs_bufferSizeExt(tridigpu_pivoting_t p, int N, int n_r, int n, size_t*
    buffer_size_in_bytes)

tridigpu_status_t
tridigpu<t>bgtfs(cudaStream_t stream, tridigpu_pivoting_t   p,
                 int          N, int         n_r, int         n,
                 const <T>*   a, const <T>* b, const <T>*   c,
                 <T>*         X, const <T>* D,
                 const void*  p_factorization_buffer, void* p_buffer)
```

The function does not allocate any extra storage, enqueues the calculation into the CUDA stream, and is non-blocking, i.e., it may return control back to the host before the result is ready.

[1] Analogously, LAPACK provides gttrf and gttrs.

Input	Description
p	recommended value: TRIDIGPU_DEFAULT_PIVOTING (see above)
N	size of the block tridiagonal system
n_r	number of right-hand sides
n	size of one block; allowed values are 1, 2, 3, 4
a	lower band of factorized tridiagonal matrix (length N * n * n)
b	diagonal of factorized tridiagonal matrix (length N * n * n)
c	upper diagonal of factorized tridiagonal matrix (length N * n * n)
D	dense right-hand side matrix and buffer of length N * n_r * n
p_factorization_buffer	buffer with additional data of the factorized matrix
p_buffer	temporarily required memory buffer on the GPU

Output	Description
X	dense solution matrix and buffer of length N * n_r * n. Buffer D may be passed also as X, then the solution replaces the right-hand side data after kernel execution.

Example:

```
size_t factorization_buffer_size_in_bytes = 0;
if (tridigpuSbgtf_factorizationBufferSizeExt(p, N, n_r, n,
    &factorization_buffer_size_in_bytes)) {
  printf("determining factorization buffer size failed\n");
}
void* p_factorization_buffer;
if (cudaMalloc(&p_factorization_buffer, factorization_buffer_size_in_bytes)) {
  printf("allocating factorization buffer failed\n");
}
if (tridigpuSbgtf(stream, p, N, n_r, n, a, b, c, p_factorization_buffer)) {
  printf("factorizing block tridiagonal system failed\n");
}
size_t buffer_size_in_bytes = 0;
if (tridigpuSbgtfs_bufferSizeExt(p, N, n_r, n, &buffer_size_in_bytes)) {
  printf("determining temporary buffer size failed\n");
}
void* p_buffer;
if (cudaMalloc(&p_buffer, buffer_size_in_bytes)) {
  printf("allocating temporary buffer failed\n");
}

while(...) {
  // - set right-hand sides `D`
  // - `p_buffer` does not need to be initialized before the call to `tridigpuSbgtfs`
  // - `p_factorization_buffer`,`a`,`b`,`c` is written by `tridigpuSbgtf` and read
    by `tridigpuSbgtfs`
  if (tridigpuSbgtfs(stream, p, N, n_r, n, a, b, c, X, D, p_factorization_buffer,
    p_buffer)) {
    printf("solving block tridiagonal system with factorization failed\n");
  }
}
```

Solve banded systems with multiple right-hand sides

This function solves Problem Class DIA.

```
tridigpu_status_t
tridigpu<t>gbsv_bufferSizeExt(tridigpu_pivoting_t p, int N, int n_r, int kl, int ku,
    size_t* buffer_size_in_bytes)

tridigpu_status_t
tridigpu<t>gbsv(cudaStream_t stream, tridigpu_pivoting_t       p,
                int          N, int          n_r, int          kl, int          ku,
                const <T>*   AB,
                <T>*         X, const <T>*   D,
                void*        p_buffer)
```

The function does not allocate any extra storage, enqueues the calculation into the CUDA stream, and is non-blocking, i.e., it may return control back to the host before the result is ready. The numeric behaviour of this function is equal to `tridigpu<t>bgtsv` and the banded system is converted on-chip into a block tridiagonal system.

Input	Description
`p`	recommended value: `TRIDIGPU_DEFAULT_PIVOTING` (see above)
`N`	size of the banded system
`n_r`	number of right-hand sides
`kl`	number of lower bands in `AB`; `kl=1,2,3,4`
`ku`	number of upper bands in `AB`; `ku=1,2,3,4`
`AB`	banded storage format of matrix and buffer of length `N * (kl + ku + 1)`
`D`	dense right-hand side matrix and buffer of length `N * n_r`
`p_buffer`	temporarily required memory buffer on the GPU

Output	Description
`X`	dense solution matrix and buffer of length `N * n_r`. Buffer `D` may be passed also as `X`, then the solution replaces the right-hand side data after kernel execution.

Example:

```
size_t buffer_size_in_bytes = 0;
if (tridigpuSgbsv_bufferSizeExt(p, N, n_r, ku, ku, &buffer_size_in_bytes)) {
  printf("determining temporary buffer size failed\n");
}
void* p_buffer;
if (cudaMalloc(&p_buffer, buffer_size_in_bytes)) {
  printf("allocating temporary buffer failed\n");
}
if (tridigpuSgbsv(stream, p, N, n_r, kl, ku, AB, X, D, p_buffer)) {
  printf("solving banded system failed\n");
}
```

Solve scalar tridiagonal systems with multiple sparse right-hand sides and dense solutions

This function solves Problem Class ScalarCSC2Dense for $\mathcal{D}$ saved in a CSC format.

```
tridigpu_status_t
tridigpu<t>gtsv_csc2dense_bufferSizeExt(tridigpu_pivoting_t p, int N, int n_r,
    size_t* buffer_size_in_bytes)

tridigpu_status_t
tridigpu<t>gtsv_csc2dense(cudaStream_t  stream, tridigpu_pivoting_t    p,
                          int           N, int         n_r,
                          const <T>*    a, const <T>* b, const <T>*    c,
                          <T>*          X,
                          const <T>*    rhs_values,
                          const int*    rhs_row_indices,
                          const int*    rhs_col_ptrs,
                          int           rhs_nnz,
                          void*         p_buffer)
```

The function does not allocate any extra storage, enqueues the calculation into the CUDA stream, and is non-blocking, i.e., it may return control back to the host before the result is ready.

Input	Description
`p`	recommended value: `TRIDIGPU_DEFAULT_PIVOTING` (see above)
`N`	size of the tridiagonal system
`n_r`	number of right-hand sides
`a`	lower band of tridiagonal matrix (length `N`)
`b`	diagonal of tridiagonal matrix (length `N`)
`c`	upper diagonal of tridiagonal matrix (length `N`)
`rhs_values`	value array of sparse right-hand sides in CSC format (length `rhs_nnz`)
`rhs_row_indices`	row index array of sparse right-hand sides in CSC format (length `rhs_nnz`)
`rhs_col_ptrs`	column pointer array of sparse right-hand sides in CSC format (length `n_r + 1`)
`rhs_nnz`	number of non-zero entries of sparse right-hand side matrix $\mathcal{D}$
`p_buffer`	temporarily required memory buffer on the GPU

Output	Description
`X`	dense solution matrix and buffer of length `N * n_r`

Example:

```
size_t buffer_size_in_bytes = 0;
if (tridigpuSgtsv_csc2dense_bufferSizeExt(p, N, n_r, &buffer_size_in_bytes)) {
  printf("determining temporary buffer size failed\n");
}
void* p_buffer;
if (cudaMalloc(&p_buffer, buffer_size_in_bytes)) {
  printf("allocating temporary buffer failed\n");
}
if (tridigpuSgtsv_csc2dense(stream, p, N, n_r, a, b, c, X, rhs_values,
    rhs_row_indices, rhs_col_ptrs, rhs_nnz, , p_buffer)) {
  printf("solving tridiagonal system with sparse rhs failed\n");
}
```

General tridiagonal solve for many sparse right-hand sides and sparse solutions

This function solves Problem Class ScalarCSC2CSC, i.e., it calculates $A^{-1}\mathcal{D}$ for a general tridiagonal matrix A and a sparse matrix $\mathcal{D}$. For large sparse matrices $\mathcal{D}$ the dense solution matrix would exceed the available memory, therefore, this functions outputs only a sparse result matrix $\mathcal{X}$. In fact, $\mathcal{X} = \texttt{prune}(A^{-1}\mathcal{D}, S_p)$, thus the result is equal to the dense result pruned to a sparsity pattern S_p. S_p is determined such that the maximum absolute values in the dense result are included in the sparse result. If $\texttt{nnz}_j$ is the number of non-zero entries in column j of matrix $\mathcal{X}$, and for each j let $\sigma_j : [0, N-1] \to [0, N-1]$ be an index permutation such that the j-column $X_{\sigma_j(i),j}$ of $X := A^{-1}\mathcal{D}$ is sorted in descending order with respect to the absolute values, then S_p is obtained by

$$(S_p)_{i,j} = \begin{cases} 1 & \text{if } \sigma_j(i) < \texttt{nnz}_j \\ 0 & \text{otherwise.} \end{cases} \tag{81}$$

In words: the sparsity pattern of $\mathcal{X}$ is controlled by the user given column pointers ($\texttt{nnz}_j$), and each column of the sparse result is filled with the maximum values of the dense result with respect to their absolute value.

```
tridigpu_status_t
tridigpu<t>gtsv_csc2csc(tridigpu_pivoting_t p,
    int         N,
    const <T>*  a, const <T>*            b, const <T>*              c,
    const <T>*  rhs_values, const int*   rhs_row_indices, const int*
    rhs_col_ptrs,
    int         n_r, int                 rhs_nnz,
    <T>*        result_values, int*      result_row_indices,
    const int*  result_col_ptrs, int     result_nnz)
```

The function allocates extra GPU storage, enqueues the calculation into the default CUDA stream, and is blocking the host, i.e., it returns control back to the host when the result is ready.

Input	Description
p	recommended value: TRIDIGPU_DEFAULT_PIVOTING (see above)
N	size of the tridiagonal system
a	lower band of tridiagonal matrix (length N)
b	diagonal of tridiagonal matrix (length N)
c	upper diagonal of tridiagonal matrix (length N)
rhs_values	CSC values of right-hand sides $\mathcal{D}$ (length nnz)
rhs_row_indices	CSC column indices of right-hand sides $\mathcal{D}$ (length nnz)
rhs_col_ptrs	CSC column pointers of right-hand sides $\mathcal{D}$ (length N + 1)
n_r	number of right-hand sides
rhs_nnz	number of non-zero entries of right-hand sides $\mathcal{D}$
result_col_ptrs	CSC column pointers of the result matrix (length N + 1)
result_nnz	number of non-zero entries of the result matrix.

Output	Description
result_values	CSC values of sparse solution matrix $\mathcal{X}$ (length result_nnz)
result_row_indices	CSC row indices of sparse solution matrix $\mathcal{X}$ (length result_nnz)