Preconditioned Krylov solvers on GPUs

Hartwig Anzt[a,*], Mark Gates[a], Jack Dongarra[a,b,c], Moritz Kreutzer[d], Gerhard Wellein[d], Martin Köhler[e]

[a] *Department for Electrical Engineering and Computer Science, University of Tennessee, Innovative Computing Lab Knoxville, Tennessee 37996, United States*
[b] *University of Manchester, UK*
[c] *Oak Ridge National Laboratory, USA*
[d] *Friedrich-Alexander University of Nuermberg-Erlangen, Germany*
[e] *Max Planck Institute for Dynamics of Complex Technical Systems Magdeburg, Germany*

A B S T R A C T

In this paper, we study the effect of enhancing GPU-accelerated Krylov solvers with preconditioners. We consider the BiCGSTAB, CGS, QMR, and IDR(s) Krylov solvers. For a large set of test matrices, we assess the impact of Jacobi and incomplete factorization preconditioning on the solvers' numerical stability and time-to-solution performance. We also analyze how the use of a preconditioner impacts the choice of the fastest solver.

Keywords:
Krylov solvers
Preconditioning
GPU
ILU
Jacobi

1. Introduction

Krylov methods are a popular choice for iteratively solving large, sparse linear systems. Their convergence properties are often superior compared to component-wise relaxation methods, and their ability to benefit from preconditioning make them attractive from the theoretical point of view. Significant efforts are spent on deriving Krylov solvers optimized for specific matrix properties, with the Conjugate Gradient (CG [1]) algorithm, suitable for symmetric positive definite problems, being the most popular example. At the same time, their generic construction as a combination of sparse matrix vector products, vector operations, and reductions makes Krylov methods attractive for implementation on parallel hardware architectures such as graphics processing units (GPUs). As a result, linear algebra software libraries like cuSPARSE, MAGMA-sparse, Paralution, or ViennaCL offer a large variety of Krylov solvers to users [2–5].

In a recent effort, we compared different Krylov methods with respect to numerical stability, convergence rate, and time-to-solution performance [6]. The motivation for this comparison is when a problem has unknown characteristics, making an

* Corresponding author.
E-mail address: hanzt@icl.utk.edu (H. Anzt).

a priori choice of the optimal solver impossible. Although the study was exclusively based on the GPU implementations of Krylov methods in the MAGMA-sparse software library, the convergence and performance results are expected to carry beyond this specific choice of software library and manycore architecture. We define a method as *robust*, in terms of numerical stability, if it converges to a small residual for a wide variety of problems. A key observation in [6] is that the Induced Dimension Reduction (IDR [7]) algorithm works very well for a large fraction of the problems: it combines a high degree of robustness with respect to problem characteristics with a time-to-solution performance often surpassing the other Krylov methods included in the study (CGS, BiCGSTAB, and QMR).

In this paper we go beyond assessing the convergence and performance of unpreconditioned Krylov methods by enhancing a selection of Krylov methods with different preconditioning techniques, as reviewed in Section 2. Our test framework, described in Section 3, uses a wide variety of matrices from the University of Florida Sparse Matrix Collection, along with solvers in the MAGMA-sparse library. Our key contribution is the comprehensive analysis in Section 4 assessing the impact of choosing a preconditioner on the metrics quantifying the suitability and superiority of the methods. Although in this paper we target only a GPU hardware setting, the key findings summarized in Section 5 are meaningful also for other manycore architectures. Similar to [6], we assume no a priori knowledge about matrix characteristics, that is, we follow a "black box" approach. This scenario reflects the situation where we look for a good method to solve a single and isolated linear system without knowing about the system characteristics.

2. Background

2.1. Related work

Designing Krylov methods that are efficient in solving non-symmetric systems is an active field of research. A survey on Krylov solvers developed before 1991 can be found in [8]. Since then, a large number of new methods have been developed, often arising as a combination of preexisting algorithms, with improved convergence and stability properties for certain problem classes. A comprehensive overview of recent Krylov methods, their algorithmic design and mathematical properties is given in [9]. For problems with an origin in partial differential equations, Saad [1] outlines the complete simulation path: from the partial differential equations with their numerical properties, via the discretization methods generating linear systems of equations, to the efficiency of Krylov methods when solving these.

Unfortunately, however, there does not exist an overall best Krylov solver, as for each method, it is possible to find a problem class where a different solver is superior [10]. The growing variety of solvers to choose from presents the challenge of selecting a suitable method. One possible workaround is to base the selection on a theoretical analysis, matching problem characteristics to algorithm properties. This requires thorough knowledge of the mathematical theory. However, a deep theoretical analysis may only be justified if a significant amount of work is spent in the solution process, e.g., if an application requires solving a sequence of linear systems with the same characteristics. If the origin of the problem and some key characteristics are known, a different approach is to base the selection on empirical knowledge. There exist multiple studies comparing the performance of different Krylov methods for specific problems [11–14].

In a "black box scenario," there is no information about the problem characteristics available to allow the use of a theoretical analysis. The only goal is to make a good guess for this single run. One approach to overcoming the challenge of choosing the right solver is the Lighthouse framework [15], which is an automated selection tool that uses a searchable taxonomy to match specific problems to algorithms. In [16], the developers of Lighthouse study how the Lighthouse framework succeeds in classifying the preconditioned iterative linear solvers in the Parallel Extensible Toolkit for Scientific Computation (PETSc) [17] and Trilinos [18] libraries.

A more conservative approach is to assess the performance of the solver candidates for a large set of test matrices. In [6], we compared a set of GPU-accelerated Krylov solvers available in the MAGMA-sparse software module with respect to their convergence and time-to-solution performance using matrices from the University of Florida Sparse Matrix Collection [19]. In this paper, we extend the previous analysis by enhancing the solvers with preconditioners, and investigate how preconditioning impacts the choice of a suitable Krylov solver.

2.2. General Krylov solvers attractive for GPU implementation

Among the most popular Krylov solvers for general systems are GMRES, BiCG, BiCGSTAB, CGS, QMR, TFQMR, and IDR(s). Except for GMRES, all these are based on a short recurrence formulation [20]. This means that in each iteration, only a few of the latest basis vectors are required to generate the new basis vector. GMRES requires storing all previous basis vectors, greatly increasing the computational cost and memory footprint. While restarted GMRES limits the number of basis vectors, it still has higher costs than other solvers – especially problematic in the limited memory environment of GPUs – and the restart parameter, defining the maximum size of the Krylov space before restarting, should be tuned for the individual problem, since the convergence rate can be sensitive to the restart parameter. While a small matrix may allow GMRES to run in a non-restarted fashion, restarting is inevitable when addressing large problems where the GPU memory limits the number of search directions that can be stored. Given this background, we did not find GMRES suitable as a black box solver for targeting a diverse set of problems, so we excluded it from our experiments.

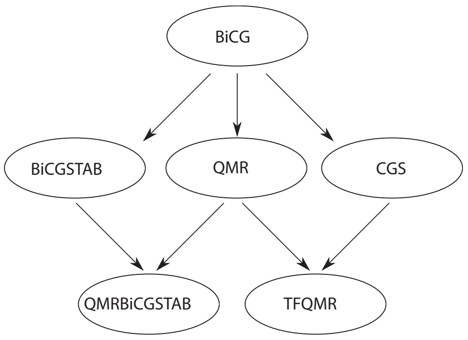

Fig. 1. Outlining dependencies between some Krylov methods based on a Bi-Lanczos process.

BiCG arises as a non-symmetric variant of the CG algorithm for symmetric positive definite (SPD) systems [1]. BiCG, however, carries some unattractive properties: it requires multiplication with the transpose of the system matrix, often has non-smooth convergence (local residual norm increase), and does not implicitly compute a residual norm estimate in the iterations that allows checking convergence. In terms of stability, BiCG suffers from two potential breakdown scenarios: pivot breakdown and, in case of a non-symmetric system matrix, Lanczos breakdown [21]. This has motivated efforts to modify the BiCG method in favor of desirable properties: avoiding breakdown, avoiding use of the transpose, smooth convergence, and an implicit residual. An effort to avoid pivot breakdown is the "quasi-minimal residual" (QMR) method developed by Freud et al. [22]. QMR does not resolve the Lanczos breakdown that can occur for non-symmetric systems. Avoiding Lanczos breakdown is much more challenging, and although numerous research efforts exist [23,24], most of the proposed ideas are not very popular in the scientific computing community. Much more successful was the search for a workaround avoiding the transposed system matrix. Sonneveld developed the very efficient "conjugate gradient square" algorithm (CGS, [25]). Squaring the BiCG polynomial removes the need for the transposed system matrix and, for linear systems where BiCG converges, CGS is often an attractive, faster alternative. Unfortunately, CGS inherits the non-smooth convergence properties and breakdown conditions from BiCG [25]. For enhanced numerical stability and smoother convergence, van der Vorst developed the BiCGSTAB method [26]. This algorithm can be seen as a combination of BiCG and GMRES using the restart parameter 1 [27]. BiCGSTAB offers an attractive balance between numerical stability and fast convergence. There are also efforts to combine QMR, CGS and BiCGSTAB to obtain a method sharing multiple enhancements; see TFQMR as a combination of QMR and CGS [28] and QMRBiCGSTAB as a combination of QMR and BiCGSTAB [29]. The relation between these solvers, all based on a Bi-Lanczos process generating the Krylov space, is diagrammed in Fig. 1.

A much more recent Krylov solver is the "induced dimension reduction" algorithm with a flexible shadow space dimension (IDR(s)) developed by Sonneveld et al. [30]. For different shadow space dimensions s, IDR(s) is a robust and efficient short recurrence Krylov subspace method. At the same time it can be seen as a generalization of different Krylov solvers, as it can be shown that for the shadow space dimension $s = 1$, IDR(1) becomes very similar to BiCGSTAB [31]. Precisely, the IDR method finds residuals in a sequence of shrinking subspaces that are orthogonal to a sequence of growing Krylov subspaces, the so-called "shadow spaces". The shadow space dimension s defines the number of randomly-chosen orthogonal vectors that are used as starting points for generating the shadow spaces [31]. To limit the storage and computational cost, we limit the IDR(s) analysis in this study to the shadow space dimensions $s = 2$, $s = 4$, and $s = 8$ (which were also proposed by Sonneveld et al. [30]). A variant of IDR(s) that is not included in this study is the Ritz-IDR proposed by Simoncini et al. [32]. Motivated by the possibility to regard IDR(s) as a Petrov–Galerkin method, Ritz-IDR uses the poles of the rational function as Ritz values [32]. The convergence of Ritz-IDR is competitive and, for small shadow space dimensions s, often superior to the initial algorithm proposed in Sonneveld and van Gijzen [30].

All presented Krylov methods are composed of a combination of sparse matrix vector products, vector updates, and dot products. The similarity in design implies that these methods are very similar in terms of parallel efficiency on GPU hardware, and they can all be implemented by using central building blocks for sparse linear algebra. We provide in Section 3 details about the efficient GPU-implementation of the considered methods.

2.3. Preconditioning Krylov methods

As previously elaborated, Krylov solvers are among the most powerful methods for iteratively solving linear systems of equations. In many cases Krylov solvers can benefit from combining with preconditioning techniques. The underlying concept is to improve the conditioning of the target problem by transforming the original linear system $Ax = b$ into the system $MAx = Mb$ [1] where the condition number $\text{cond}_2(MA)$ is much smaller than $\text{cond}_2(A)$. Addressing a problem with better conditioning is typically easier for the iterative solver. As a result, a sophisticated preconditioner often improves both the numerical stability and the convergence rate of the iterative solver.

However, a preconditioner is only attractive if the provided convergence improvement compensates for the additional work: the preconditioner generation and the preconditioner application in the solver iterations.

In the context of High Performance Computing (HPC), this is strongly related to the question of how efficient a preconditioner can be computed and applied on a parallel architecture. A Jacobi preconditioner scales the linear system with the inverse of the diagonal of the system matrix, and therefore allows for good parallelization [1]. At the same time, the convergence benefit of Jacobi preconditioning is often limited, and typically much smaller than a more sophisticated preconditioner like an incomplete LU factorization (ILU [1]). These approximate the factorization of the system matrix on some nonzero pattern, and use sparse triangular solves in the distinct iteration steps for transforming the original problem into the preconditioned one. ILU preconditioners are robust and often succeed in significantly improving the convergence of an iterative solver. Unfortunately, the convergence improvement of ILU preconditioning comes at the price of difficult-to-parallelize sparse triangular solves that can become a bottleneck on parallel architectures. Level-scheduling [22,33] and multicolor-ordering [34,35] techniques can be used to parallelize the sparse triangular solves. However, the potential of these parallelization strategies typically decreases with the fill-in allowed in the incomplete factorization process. To balance between convergence improvement and parallel execution efficiency, we focus in this paper on the ILU(0) incomplete factorization preconditioner that ignores any fill-in [1]. Beside this configuration, we assess the performance of the unpreconditioned Krylov solvers and the Jacobi preconditioned versions.

3. Test framework

3.1. Test matrices

For this study, we use a large set of test matrices from the University of Florida Sparse Matrix Collection (UFMC [36]). Complementary to the results presented in [6], the main focus of this paper is not on comparing the robustness of unpreconditioned Krylov solvers, but rather on assessing the performance in the context of preconditioning. Given this background, we choose a set of test matrices where a significant portion of the solver/preconditioner configurations succeed. The results presented in [6] indicate that the IDR(s) algorithm is very robust with respect to problem suitability. Using an ILU(0) preconditioner typically enhances robustness and convergence properties [37]. This motivates basing the test suite on the successful convergence of an ILU(0) preconditioned IDR(4). Specifically, the test suite is composed of all matrices available at UFMC for which an IDR(4) iterative solver preconditioned with ILU(0) converges to a relative residual norm $\|r\| \leq 10^{-10}\|b\|$ within 10,000 iterations. However, problems smaller than 100 rows, and problems where the ILU(0) preconditioned IDR(4) from MAGMA-sparse completes within 0.1 s, are considered “too easy” to require the use of a GPU solver, so are excluded. Applying these filters, we obtain a test suite containing 456 “reasonably difficult” problems. Some of these systems are symmetric positive definite (SPD). Although we recognize that for these systems, the Conjugate Gradient method is well-known to be a very efficient alternative [1], we ignore this information and handle all systems as general. This is motivated by the goal of increasing the size of the test suite. In our previous study we excluded SPD systems even though they fulfilled the convergence criteria, and allowed for higher iteration counts for large systems [6]. Hence, the test suite we use in this paper adds more “easy” problems, where the Krylov solvers converge within a few iterations, and excludes a few very long running problems.

All experiments use a right-hand side $b \equiv 1$, are started with an initial guess $x \equiv 0$, and convergence is defined as the residual fulfills $\|r\| \leq 10^{-10}\|b\|$. The Krylov methods we consider differ with respect to how many basis vectors are generated in each iteration: BiCGSTAB, CGS, and QMR all have 2 sparse matrix-vector multiplications (SpMVs) per iteration; IDR(s) has $2s+1$ SpMVs in every outer iteration. For a fair comparison, we set the upper iteration bound for the distinct solvers with respect to the SpMVcount. The upper bound for the SpMVcount is set to 1,000,000 for the unpreconditioned solvers and the Jacobi preconditioner, and to 500,000 for the ILU(0) preconditioner. The lower iteration limit for the ILU-preconditioned methods is motivated by the cost of applying an ILU(0) preconditioner typically being larger than creating an additional Krylov search direction via a sparse matrix vector multiplication: both operations are memory bound, have to access every matrix entry at least once, while the parallelism level in the preconditioner application is limited by the dependency graph specific to the matrix nonzero pattern. Although an upper iteration limit is questionable in general, there exists no practical workaround. Also, choosing these very generous upper bounds makes the limits mostly irrelevant, as almost all converging test cases require significant less iterations. All timing results include the preconditioner setup time.

3.2. The libufget library

The UFMC, where we source our test matrices, is normally interfaced by a MATLAB interface or a Java application. Both interfaces allow one to search and download matrices by their name or ID. Additionally, the Java interface has the ability to search for matrices using metadata information, but once identified each matrix has to be downloaded manually. Having our conditions from the problem setup in mind, this is cumbersome for a large matrix set. In the case of the MATLAB interface we have an even worse situation because we have to check each matrix for fulfilling the required properties after downloading it. Furthermore, both interfaces cannot be easily used from C or integrated into standalone programs.

We solved this issue by developing a C library named *libufget* which allows us to access the UFMC directly from a C program. The library downloads the metadata from the UFMC and converts it into an SQLite database. In contrast to the array of structures used by the MATLAB interface or the CSV-file (comma-separated-values) used by the Java interface, the SQLite database allows fine-grained searches for selecting matrices. Beside selecting matrices by their names, we can easily

Table 1
Efficiency of the Krylov solvers included in the MAGMA-sparse software distribution: memory transfers, achieved bandwidth, and efficiency. All Krylov solvers are enhanced with a Jacobi preconditioning (diagonal scaling), and use double precision in all computations. The column-indexes and the row-pointer are stored as 32-bit integers. The analysis is based on one iteration (BiCGSTAB, CGS, QMR) or one shadow space loop (IDR(2), IDR(4), IDR(8)).

Krylov solver	Memory transfers [Byte]	Bandwidth [GB/s]	Efficiency
Jacobi-BiCGSTAB	$8 \cdot (37n + 3nnz)$	112.47	59%
Jacobi-CGS	$8 \cdot (36n + 3nnz)$	113.82	59%
Jacobi-QMR	$8 \cdot (50n + 3nnz)$	123.49	64%
Jacobi-IDR(2)	$8 \cdot (8 * ((22 + 9 \cdot 2 + 55 \cdot 1)n + 7.5n + 4.5nnz + 15n)$	130.83	68%
Jacobi-IDR(4)	$8 \cdot (8 * ((22 + 9 \cdot 8 + 55 \cdot 2)n + 12.5n + 7.5nnz + 25n)$	135.58	71%
Jacobi-IDR(8)	$8 \cdot (8 * ((22 + 9 \cdot 32 + 55 \cdot 4)n + 22.5n + 13.5nnz + 45n)$	142.64	74%

take the dimension, number of nonzero elements, symmetry, and other meta information into account. For a list of all searchable meta information, see [36]. The search result is obtained as an iterator over all matrices matched by the query. This enables us to execute an algorithm, in our case the preconditioned iterative solver, for each matrix matched by query. In terms of SQL, we can preselect all matrices that might fulfill our conditions from Section 3.1 using the following statement:

```
SELECT * FROM matrices WHERE rows == cols && \
  rows >= 100 && isReal==1
```

and automatically check if these matrices will satisfy the desired conditions mentioned in the previous section. This avoids the manual selection of the 456 matrices out of the more than 2000 matrices in the UFMC.

In future versions of libufget we will integrate additional metadata, like the convergence information, in order simplify the realization of linear system related benchmark tasks.

3.3. MAGMA-sparse

MAGMA-sparse is a software module in the accelerator-focused linear algebra software library MAGMA, developed at the University of Tennessee. At this point, MAGMA-sparse does not support data-parallel heterogeneous computing, but executes all matrix and vector operations on the hardware accelerator. Comprehensive support for NVIDIA GPUs is provided, while some basic routines and functionalities are also available for OpenCL and the Xeon Phi. MAGMA-sparse contains a large variety of solvers, preconditioners, and eigensolvers for sparse linear systems. MAGMA-sparse implements the presented Krylov solvers by interfacing to the CSR-based sparse matrix vector product available in NVIDIA's cuSPARSE library [2]. For the vector updates and the dot products, algorithm-specific routines are used that merge multiple vector operations into a single kernel [38]. This strategy is well known to reduce the main memory access compared to composing algorithms out of the standard linear algebra building blocks [39]. To test the efficiency of our implementation, we examine the achieved memory bandwidth performance for a 5-point stencil matrix with n = 9,000,000. In Table 1 we list the memory transfers needed by the distinct implementations. Similar to the analysis conducted in [40], it can be stated that all involved compute kernels are memory bound. Precisely, the Roofline performance model [40] states that the performance of these kernels is limited by the product of the computational intensity and the maximum attainable peak memory bandwidth. In terms of runtime, this limit defines a lower bound equal to the ratio between the main memory transfer volume (V) and the attainable peak transfer rate ($bw_{\max}$). For BiCGSTAB, CGS, and QMR, we count the memory transfers in a single iteration, generating always two Krylov search direction. For the different shadow space dimensions s of IDR, we count the memory transfers needed in one shadow space loop, generating $s + 1$ Krylov search directions. We use double precision in all computations, and store the column-indexes and row-pointers needed for the SpMV using 32-bit integers. We enhance all methods with a scalar Jacobi preconditioner (diagonal scaling). Although it is possible to merge the preconditioner application or the SpMV with vector operations [41], we refrain from this optimization step to maintain the flexibility of using a different preconditioner or SpMV kernel. For this setting, the main memory transfers for one shadow space loop of IDR(s) derive as [42]:

$$V_{\mathrm{IDR}(s)} = 8 \cdot \left(\underbrace{\left(9\frac{s^2}{2} + 55\frac{s}{2} + 22\right)n}_{\text{vector updates}} + \underbrace{(1.5nnz + 2.5n) \cdot (s+1)}_{\text{SpMV}} + \underbrace{5n \cdot (s+1)}_{\text{Jacobi preconditioner}} \right).$$

Similarly, the memory transfers in one iteration of BiCGSTAB, CGS, and QMR can be derived; see Table 1. We notice that assessing the efficiency of the Krylov methods using a main memory bandwidth analysis does not account for non-coalescent memory access (e.g. in the SpMV kernel) or reduction operations like in the dot product. This favors algorithms where vector updates allowing for purely coalescent reads (axpy-like operations) have a more significant share of the computational cost, see e.g. bandwidth results for QMR, and IDR(s) using larger shadow space dimensions. Using a peak bandwidth of $bw_{\max} = 192$ GB/s, which is what we achieved in bandwidth benchmarks on this architecture [42], we obtain efficiency levels around 60%–70% for all Krylov solver implementations. We did not include the incomplete factorization preconditioner

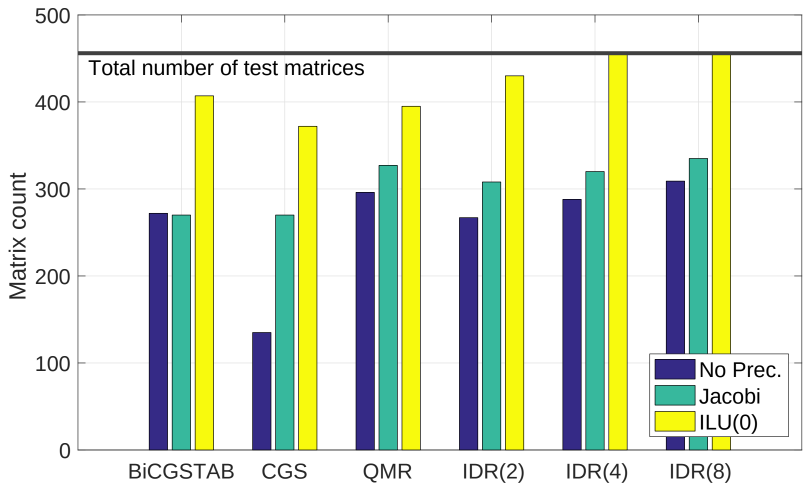

Fig. 2. Statistics on how many of the 456 problems can be solved with the different solver/preconditioner configurations.

(ILU(0)) in this analysis, as the performance of sparse triangular solves is limited by the dependency graph of the specific sparsity structure. Typically, the performance of sparse triangular solves is much lower than the memory bandwidth. MAGMA-sparse interfaces to the incomplete factorization routines available in NVIDIA's cuSPARSE library [2]. These exploit parallelism by using level-scheduling strategies, and achieve good performance for a large range of problems [43].

3.4. GPU hardware setup

The GPU target architecture is an NVIDIA Tesla K40 GPU (Kepler microarchitecture) with a theoretical peak performance of 1682 Gflop/s (double precision). The 12 GiB of GPU main memory can be accessed at a theoretical bandwidth of 288GB/s. Transfer rates around 193GB/s were reported for bandwidth experiments [42]. The Krylov solvers we use keep all matrix and vector data and most of the scalar values in the GPU memory. All vector operations are handled by the accelerator. For completeness, we nevertheless mention the host being an Intel Xeon E5 processor (Sandy Bridge). The MAGMA implementation uses CUDA and cuSPARSE version 7.5 [2]. All reported data is the mean of three experiment runs.

4. Experimental results

4.1. Enhancing GPU-accelerated Krylov solvers with preconditioning

In a first experiment, we consider the Krylov solvers separately, and assess the effect of preconditioning on the solvers' numerical stability and time-to-solution performance. Fig. 2 visualizes for how many problems each Krylov method converges, and how this number changes when enhancing the solver with a Jacobi preconditioner (diagonal scaling) or an ILU(0) preconditioner. As expected from the results in [6], the IDR(8) solver is the most robust method (in terms of numerical stability) with respect to problem characteristics, when used without a preconditioner. Using a Jacobi preconditioner significantly enhances the robustness of the CGS method. Compared to the unpreconditioned CGS, the Jacobi preconditioned CGS converges for more than twice as many problems. A Jacobi preconditioner also improves the robustness for IDR(s) and QMR, but by a smaller margin. The Jacobi preconditioned BiCGSTAB solver does not converge for more problems than the plain BiCGSTAB method.

All solvers benefit from ILU preconditioning. Equipped with the ILU(0), the IDR(4) and IDR(8) solvers converge for all problems, while IDR(2) converges for 430 of the 456 test cases. BiCGSTAB, CGS, and QMR are less robust to problem characteristics, but they also benefit from ILU preconditioning and increase their success rate by about 30%. Specifically, in combination with an ILU(0) preconditioner, BiCGSTAB, CGS, and QMR converge for 407, 372, and 395 test problems, respectively. A central observation of this data is that a Jacobi preconditioner has typically minor impact on robustness, while ILU(0) preconditioning can significantly enhance the robustness of the Krylov methods. An exception is the CGS case, where the Jacobi preconditioner also significantly improved the solver's success rate.

Although a Jacobi preconditioner fails to significantly improve the robustness of the Krylov solver, its usage can still be beneficial due to performance aspects: the diagonal scaling is embarrassingly parallel, and the potentially faster convergence can easily succeed in reducing the time-to-solution. This aspect is analyzed in Figs. 3 and 4. For the test cases where both the unpreconditioned and preconditioned solver converge, the speedup obtained from using the preconditioner is computed, and the test cases are ordered by increasing speedup. This results in two characteristic graphs, the left side comparing unpreconditioned with Jacobi preconditioning, the right side comparing unpreconditioned with ILU(0) preconditioning. The

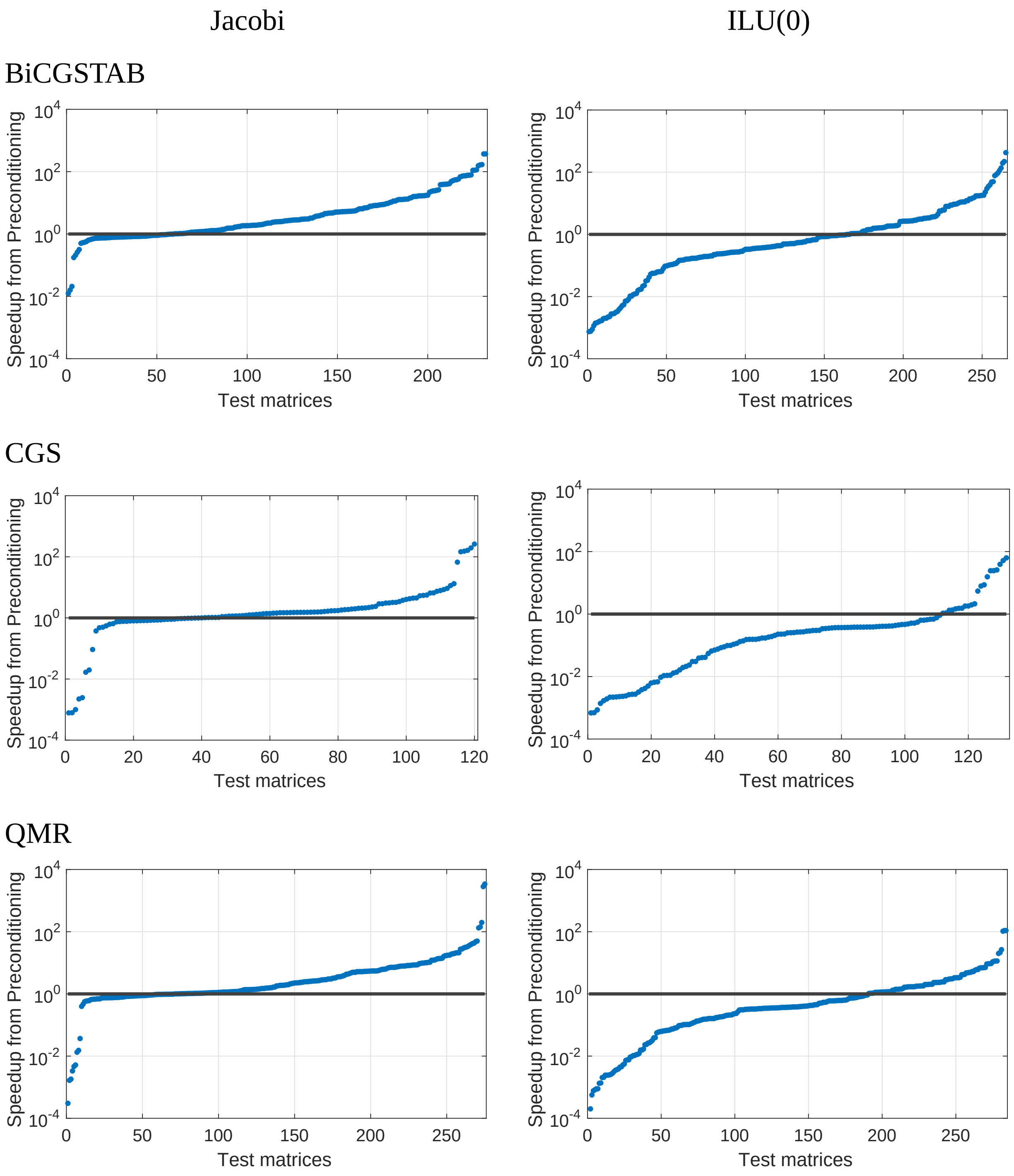

Fig. 3. Speedup obtained from using a Jacobi preconditioner (left side) or an ILU(0) preconditioner over the execution time of the unpreconditioned Krylov solver. Values smaller 1 indicate test cases where the preconditioner increased the solver execution time.

data visualized on the left side in Figs. 3 and 4 show that Jacobi preconditioning is rarely harmful to the time-to-solution performance, but can reduce the time-to-solution by up to two orders of magnitude.

The situation is very different when enhancing BiCGSTAB, CGS, or QMR with an ILU(0) preconditioner. For those test matrices where the unpreconditioned method converges, adding ILU(0) preconditioning typically increases the runtime, shown by speedups smaller than 1 on the right-hand side of Fig. 3. This comes from the fact that the ILU(0) preconditioner improves the convergence rate, but not enough to compensate for the difficult-to-parallelize sparse triangular solves in every

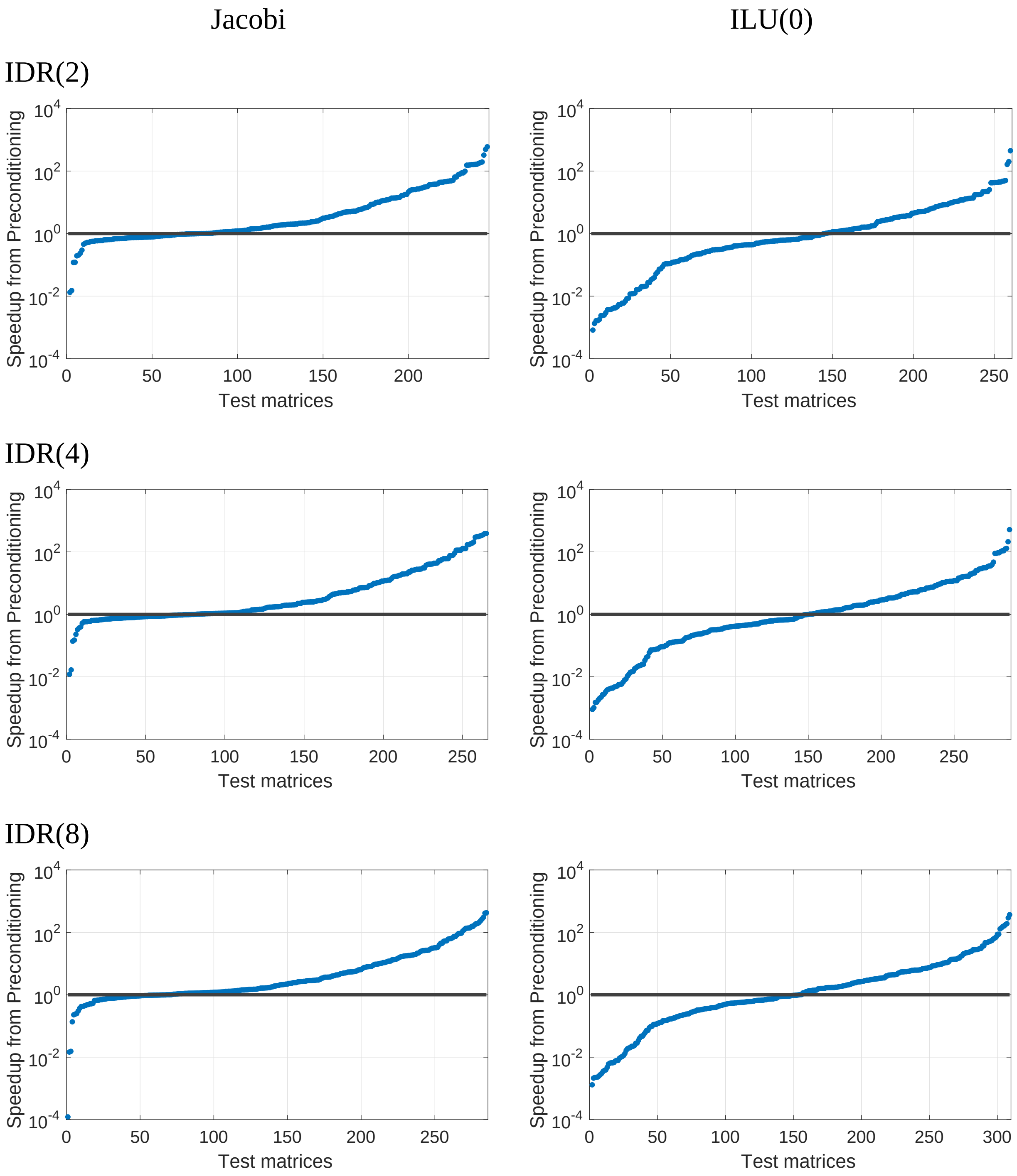

Fig. 4. Speedup obtained from using a Jacobi preconditioner (left side) or an ILU(0) preconditioner over the execution time of the unpreconditioned Krylov solver. Values smaller 1 indicate test cases where the preconditioner increased the solver execution time.

solver iteration. The iterations of the IDR(s) solver are more expensive, accumulating multiple SpMVs with the system matrix. Therefore, the impact of the expensive sparse triangular solves is damped, and more test cases benefit from ILU(0) preconditioning, as shown in right-hand side of Fig. 4. Nevertheless, when unpreconditioned IDR(s) converges, it is not clear whether ILU(0) preconditioning adds any performance benefits. Complementary to the results shown in Fig. 2, we conclude that Jacobi preconditioning typically improves performance of a GPU-accelerated Krylov method, while the performance benefits of an ILU(0) preconditioner strongly depend on the characteristics of the test case.

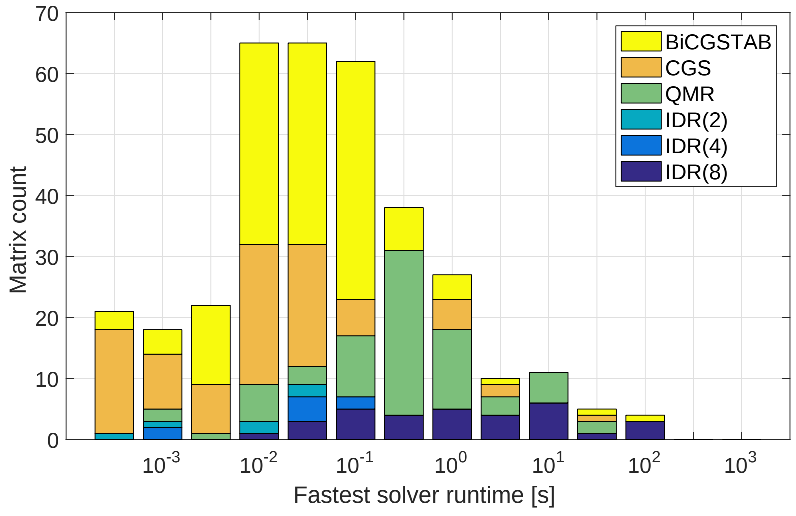

Fig. 5. Number of problems where a certain solver is the fastest vs. the runtime of the fastest solver: for each problem, the fastest time to solution determines the location on the x-axis; the color code indicates which solver is the fastest. The solvers do not use any preconditioner.

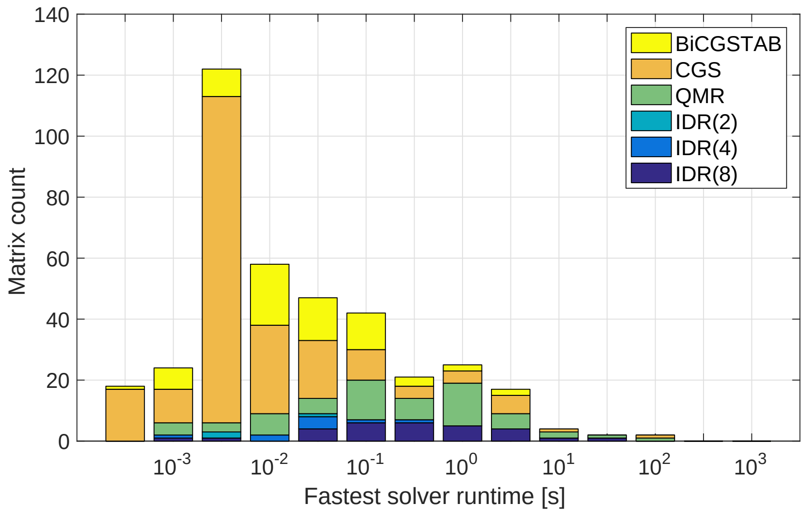

Fig. 6. Number of problems where a certain solver is the fastest vs. the runtime of the fastest solver. The solvers use a Jacobi preconditioner.

4.2. Krylov solver performance comparison

Next, we compare the different Krylov methods among themselves. We map the number of test problems that a certain solver is the performance winner to the runtime of the fastest solver in the set. In Fig. 5 we visualize the analysis for the unpreconditioned Krylov methods. The light-weight CGS solver wins "easy" test cases that have a short runtime. The BiCGSTAB solver is more robust, and suitable for "moderately difficult" problems. QMR increases the robustness further, however at the cost of more expensive iterations. The performance winner for the difficult-to-solve problems are the IDR(s) methods. As expected from the results in [6], increasing shadow space dimensions s come with increased robustness. Although the test suite was based on the successful convergence of ILU(0) preconditioned IDR(4), IDR(8) is the performance winner for the most difficult problems considered.

The Jacobi preconditioner makes the CGS method very attractive for a large number of problems, shown by Fig. 6. It combines enhanced robustness with the benefits of light-weight iterations. BiCGSTAB and QMR are still suitable for more difficult problems, while IDR(8) remains the method of choice for the hard problems. Fig. 7 visualizes how the distinct test cases migrate from the fastest unpreconditioned solvers (left-hand side) to the fastest Jacobi preconditioned solvers (right-hand side). Although the IDR(s) solver is the most robust among the methods, for most problems there exists another solver with a better time-to-solution performance. In the unpreconditioned scenario, the BiCGSTAB solver is the fastest method for many test cases. Adding a Jacobi preconditioner, many of the test cases migrate to the CGS method. Similarly, some test cases

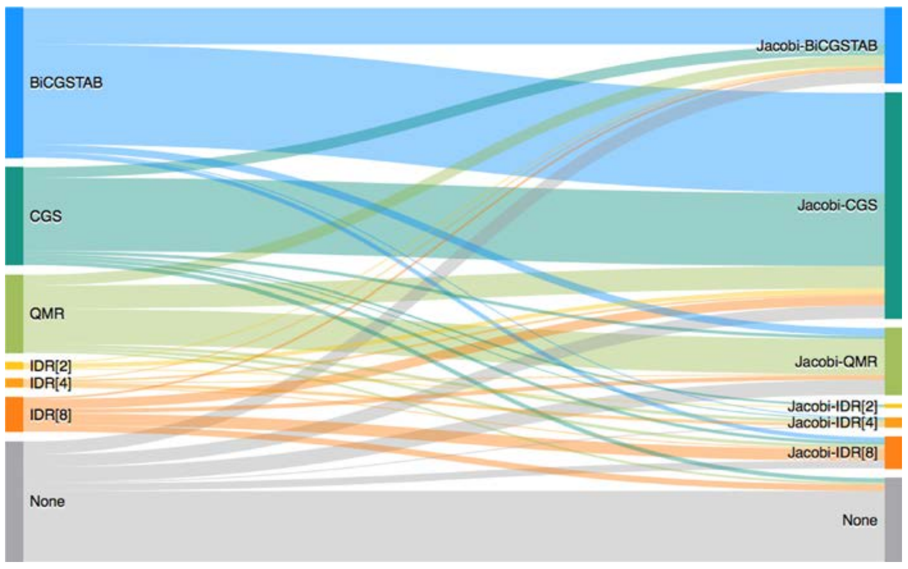

Fig. 7. Choosing the runtime-optimal solver: the height of the bars indicates for how many of the 456 test matrices a certain configuration is the fastest. The solvers on the left do not use any preconditioner; the results on the right use a Jacobi preconditioner.

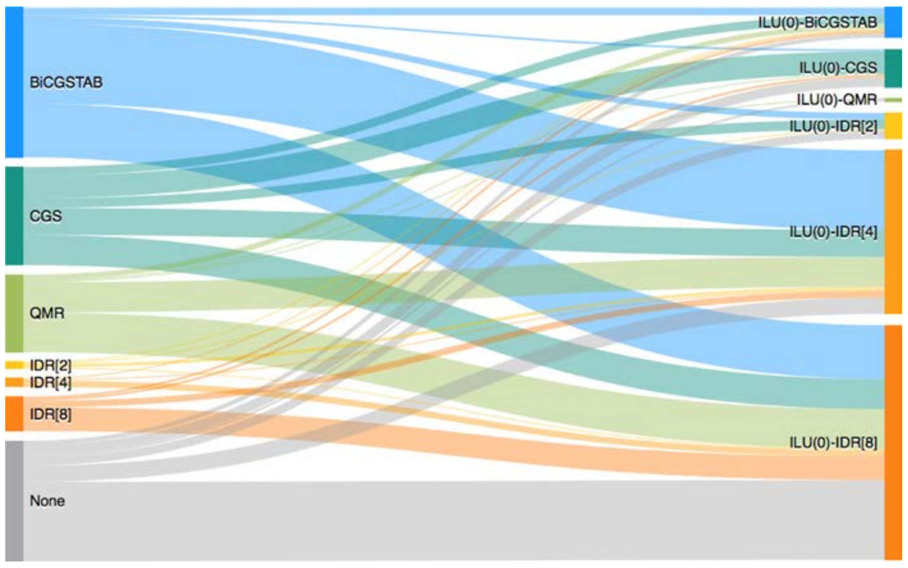

Fig. 8. Choosing the runtime-optimal solver: the height of the bars indicates for how many of the 456 test matrices a certain configuration is the fastest. The solvers on the left do not use any preconditioner; the results on the right use an ILU(0) preconditioner. "None" means that no solver converged within the iteration limit.

where QMR is the fastest solver in a unpreconditioned setting are solved faster by CGS when using Jacobi preconditioning. Relating this observation to the statistics in Fig. 2, the Jacobi preconditioner succeeds in adding the robustness needed for convergence to the inherently fast CGS solver. Adding a Jacobi preconditioner decreases the number of problems that cannot be solved by any of the methods considered. Generally, the diagonal scaling of a Jacobi preconditioner should not destroy convergence, the few exceptions represent rounding effects.

Fig. 8 shows the migration graph for enhancing the unpreconditioned solvers with an ILU(0) preconditioner. The results on the left-hand side are identical to the left-hand side in Fig. 7, while the right-hand side is radically different: the hard-to-parallelize sparse triangular solves become the performance bottleneck and, in many cases, the computationally most expensive part of the iterations. This makes methods requiring fewer iterations to converge more attractive. The share of test problems the BiCGSTAB and the CGS method are the fastest solver are very small for ILU(0) preconditioning. The share of the ILU(0) preconditioned QMR is almost negligible. This is partly related to the fact that preconditioning the QMR method requires also generating the transpose preconditioner [1], making the preconditioner setup more expensive.

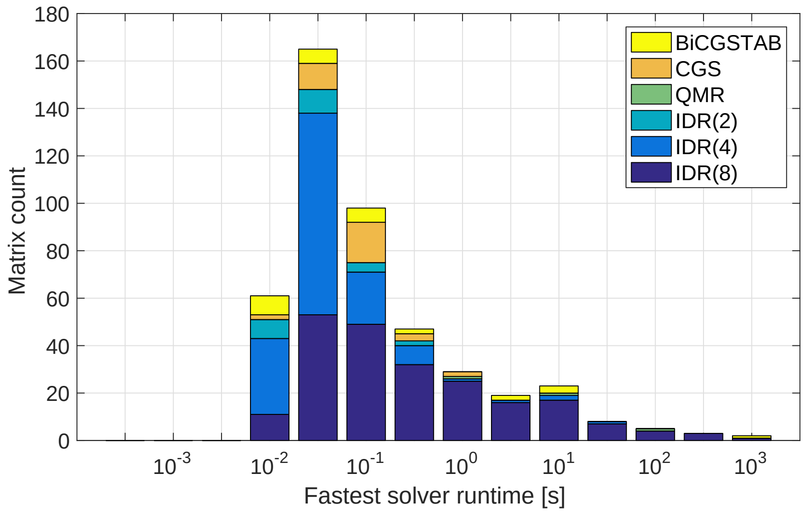

Fig. 9. Number of problems where a certain solver is the fastest vs. the runtime of the fastest solver. The solvers use an ILU(0) preconditioner.

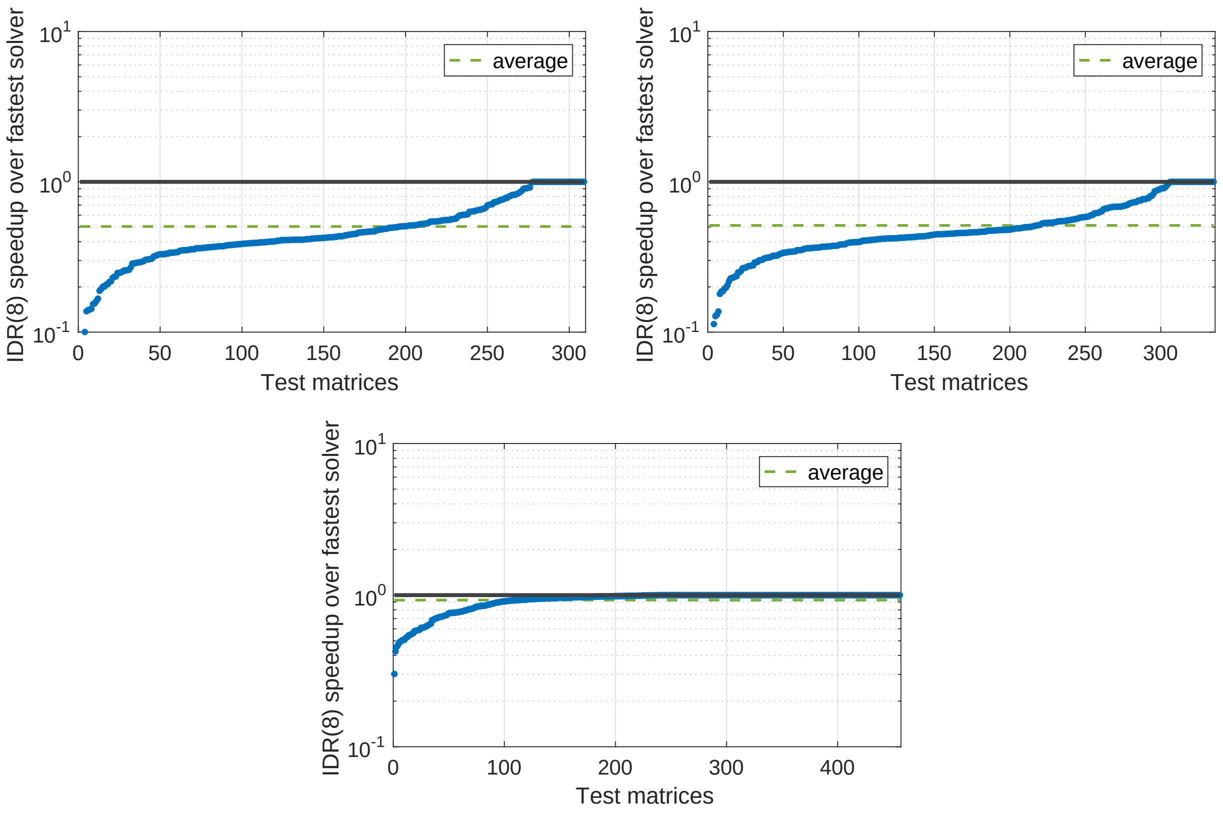

Fig. 10. Speedup obtained from using IDR(8) instead of the fastest converging solver. On the left side of the top row, the solvers do not use any preconditioner, on the right side the solvers are enhanced with a Jacobi preconditioner. The bottom row is the ILU(0) preconditioned setting. Values smaller 1 indicate test cases where using IDR(8) increases the solver execution time.

Overall, IDR(s) is the fastest method when using an ILU(0) preconditioner. Fig. 9 shows that larger shadow space dimensions s should be preferred for challenging problems: despite the fact we based the test matrix suite on the convergence of ILU(0) preconditioned IDR(4), ILU(0) preconditioned IDR(8) is the fastest solver for the “difficult” problems.

Having identified IDR(8) as typically being the fastest solver in an ILU preconditioned setting, Fig. 10 quantifies the drawback of choosing IDR(8) in a setting where a different Krylov method converges faster. For each test case, the speedup of IDR(8) over the fastest converging solver is computed, and the values are visualized in increasing order. A speedup of 1 reflects the case for IDR(8) being the fastest solver, while a “speedup” less than 1 reflects IDR(8) being slower than the (a priori unknown) fastest solver. In the graph on the left-hand side of Fig. 10, the solvers are used without a preconditioner, on the right-hand side a Jacobi preconditioner is used. In both cases, about 80% of the test problems benefit from using a different solver. For some problems, IDR(8) is up to an order of magnitude slower than the fastest solver. However, on average, the speedup is about 0.5, which is consistent with the observation in [6] of IDR(8) being on average about twice slower than the fastest solver.

5. Summary

In this paper we have investigated the effect of preconditioning GPU-accelerated Krylov solvers. For a large set of test matrices, we have analyzed the effect of Jacobi- and ILU preconditioning on robustness and time-to-solution performance for the BiCGSTAB, CGS, QMR, and IDR(s) Krylov solvers. We have demonstrated that regarding performance aspects, the GPU-accelerated Krylov solvers typically benefit from using the Jacobi preconditioner (embarrassingly parallel diagonal scaling). Incomplete factorization preconditioning improves the robustness of the solvers, but the difficult-to-parallelize sparse triangular solves can become a bottleneck increasing the overall solver runtime. In ILU preconditioned settings, IDR(s) is attractive as the preconditioner applications have a smaller share in the total iteration cost. Using a Jacobi preconditioner, there typically exists a faster solver, but IDR(8) is on average only twice slower than the (unknown) optimal choice. This good balance between robustness and performance makes IDR(8) – especially when enhanced with Jacobi- or ILU preconditioning – an attractive choice in a black box scenario where no information about the optimal solver is available. We emphasize that these findings are expected to carry beyond the specific architecture we used, as the implementations of the distinct methods share very similar memory performance.

References

[1] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
[2] NVIDIA Corporation CUDA Toolkit v7.5, September,, 2015.
[3] Innovative Computing Lab, MAGMA version 2.0, 2016, (http://icl.cs.utk.edu/magma/).
[4] MAGMA, PARALUTION, 2015, (http://www.paralution.com/).
[5] ViennaCL, 2015, (http://viennacl.sourceforge.net/).
[6] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, M. Koehler, Efficiency of general Krylov methods on GPUs – an experimental study, in: 2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 683–691.
[7] H. Anzt, E. Ponce, G.D. Peterson, J. Dongarra, Gpu-accelerated co-design of induced dimension reduction: Algorithmic fusion and kernel overlap, in: Proceedings of the 2nd International Workshop on Hardware-Software Co-Design for High Performance Computing, Co-HPC ’15, ACM, New York, NY, USA, 2015, pp. 5:1–5:8.
[8] R.W. Freund, G.H. Golub, N.M. Nachtigal, Iterative solution of linear systems, Acta Numerica 1 (1992) 57–100.
[9] V. Simoncini, D.B. Szyld, Recent computational developments in Krylov subspace methods for linear systems, Numer. Linear Algebra Appl. 14 (1) (2007) 1–59, doi:10.1002/nla.499.
[10] N.M. Nachtigal, S.C. Reddy, L.N. Trefethen, How fast are nonsymmetric matrix iterations? SIAM J. Matrix Anal. Appl. 13 (3) (1992) 778–795.
[11] L. Giraud, D. Ruiz, A. Touhami, A comparative study of iterative solvers exploiting spectral information for SPD systems, SIAM J. Sci. Comput. 27 (5) (2006) 1760–1786.
[12] F. Perini, E. Galligani, R.D. Reitz, A study of direct and Krylov iterative sparse solver techniques to approach linear scaling of the integration of chemical kinetics with detailed combustion mechanisms, Combust. Flame 161 (5) (2014) 1180–1195.
[13] S. Sundar, B. Bhagavan, Comparison of Krylov subspace methods with preconditioning techniques for solving boundary value problems, Comput. Math. Appl. 38 (11–12) (1999) 197–206.
[14] D.C. Fong, M. Saunders, CG versus MINRES: An Empirical Comparison, Technical Report, Stanford University, 2011.
[15] The Lighthouse Framework, 2016, (http://lighthousehpc.github.io/lighthouse/).
[16] P. Motter, K. Sood, E. Jessup, B. Norris, Lighthouse: an automated solver selection tool, in: Proceedings of the 3rd International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering, SE-HPCCSE ’15, ACM, New York, NY, USA, 2015, pp. 16–24, doi:10.1145/2830168.2830169.
[17] S. Balay, S. Abhyankar, M.F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W.D. Gropp, D. Kaushik, M.G. Knepley, L.C. McInnes, K. Rupp, B.F. Smith, S. Zampini, H. Zhang, H. Zhang PETSc 2016,. http://www.mcs.anl.gov/petsc.
[18] M.A. Heroux, R.A. Bartlett, V.E. Howle, R.J. Hoekstra, J.J. Hu, T.G. Kolda, R.B. Lehoucq, K.R. Long, R.P. Pawlowski, E.T. Phipps, A.G. Salinger, H.K. Thornquist, R.S. Tuminaro, J.M. Willenbring, A. Williams, K.S. Stanley, An overview of the trilinos project, ACM Trans. Math. Softw. 31 (3) (2005) 397–423, doi:10.1145/1089014.1089021.
[19] T.A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25, doi:10.1145/2049662.2049663.
[20] J. Liesen, Z. Strakoš, On optimal short recurrences for generating orthogonal Krylov subspace bases, SIAM Rev. 50 (3) (2008) 485–503.
[21] C. Brezinski, M. Redivo-Zaglia, H. Sadok, Breakdowns in the implementation of the lánczos method for solving linear systems, Comput. Math. Appl. 33 (1–2) (1997) 31–44.
[22] R.W. Freund, N.M. Nachtigal, QMR: A quasi-minimal residual method for non-hermitian linear systems, Numerische Mathematik 60(1) 315–339.
[23] C. Brezinski, M.R. Zaglia, H. Sadok, Avoiding breakdown and near-breakdown in lanczos type algorithms, Numer. Algorithms 1 (2) (1991) 261–284.
[24] C. Brezinski, M. Redivo-Zaglia, Treatment of near-breakdown in the CGS algorithm, Numer. Algorithms 7 (1) (1994) 33–37.
[25] P. Sonneveld, CGS, a fast Lanczos-type solver for nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 10 (1) (1989) 36–52.
[26] H.A. van der Vorst, Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 13 (2) (1992) 631–644.
[27] M.H. Gutknecht, Variants of BICGSTAB for matrices with complex spectrum, SIAM J. Sci. Comput. 14 (5) (1993) 1020–1033.
[28] R.W. Freund, A transpose-free quasi-minimal residual algorithm for non-hermitian linear systems, SIAM J. Sci. Comput. 14 (2) (1993) 470–482.
[29] T.F. Chan, E. Gallopoulos, V. Simoncini, T. Szeto, C.H. Tong, A quasi-minimal residual variant of the bi-CGSTAB algorithm for nonsymmetric systems, SIAM J. Sci. Comput. 15 (2) (1994) 338–347.
[30] P. Sonneveld, M.B. van Gijzen, IDR(s): a family of simple and fast algorithms for solving large nonsymmetric systems of linear equations, SIAM J. Sci. Comput. 31 (2) (2009) 1035–1062, doi:10.1137/070685804.
[31] G.L. Sleijpen, P. Sonneveld, M.B. van Gijzen, Bi-CGSTAB as an induced dimension reduction method, Appl. Numer. Math. 60 (11) (2010) 1100–1114. Special Issue: 9th {IMACS} International Symposium on Iterative Methods in Scientific Computing (IISIMSC 2008)
[32] V. Simoncini, D.B. Szyld, Interpreting IDR as a Petrov-Galerkin method, SIAM J. Sci. Comput. (2010) 1898–1912.

[33] Y. Saad, Multilevel ILU with reorderings for diagonal dominance, SIAM J. Sci. Comput. 27 (3) (2005) 1032–1057.
[34] D. Lukarski, Parallel Sparse Linear Algebra for Multi-core and Many-core Platforms – Parallel Solvers and Preconditioners, Ph.D. thesis. Karlsruhe Institute of Technology, 2012.
[35] M. Benzi, W. Joubert, M. G., Numerical experiments with parallel orderings for ILU preconditioners, Electron. Trans. Numer. Anal. 8 (1999) 88–114.
[36] T.A. Davis, Y. Hu, The university of florida sparse matrix collection, ACM Trans. Math. Softw. 38 (1) (2011) 1–25, doi:10.1145/2049662.2049663.
[37] E. Chow, Y. Saad, Experimental study of ilu preconditioners for indefinite matrices, J. Comput. Appl. Math. 86 (2) (1997) 387–414.
[38] H. Anzt, W. Sawyer, S. Tomov, P. Luszczek, I. Yamazaki, J. Dongarra, Optimizing Krylov subspace solvers on graphics processing units, 28th IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW 2014), 2014.
[39] K. Rupp, J. Weinbub, A. Jüngel, T. Grasser, Pipelined iterative solvers with kernel fusion for graphics processing units, ACM Trans. Math. Software (TOMS) 43 (2) (2016) 11.
[40] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, doi:10.1145/1498765.1498785.
[41] H. Anzt, M. Baboulin, J. Dongarra, Y. Fournier, F. Hulsemann, A. Khabou, Y. Wang, Accelerating the conjugate gradient algorithm with GPU in CFD simulations, Vecpar 2016, 12th International Meeting on High Performance Computing for Computational Science 2th International Meeting on High Performance Computing for Computational Science, (submitted).
[42] H. Anzt, M. Kreutzer, E. Ponce, G.D. Peterson, G. Wellein, J. Dongarra, Optimization and performance evaluation of the IDR iterative krylov solver on GPUs, Int. J. High Perform. Comput. (2016), doi:10.1177/1094342016646844.
[43] M. Naumov, P. Castonguay, J. Cohen, Parallel Graph Coloring with Applications to the Incomplete-LU Factorization on the GPU, Technical Report, NVIDIA, 2015.