Iterative Sparse Triangular Solves for Preconditioning

Hartwig Anzt[1](✉), Edmond Chow[2], and Jack Dongarra[1]

[1] University of Tennessee, Knoxville, TN, USA
hanzt@icl.utk.edu, dongarra@eecs.utk.edu
[2] Georgia Institute of Technology, Atlanta, GA, USA
echow@cc.gatech.edu

Abstract. Sparse triangular solvers are typically parallelized using level-scheduling techniques, but parallel efficiency is poor on high-throughput architectures like GPUs. We propose using an iterative approach for solving sparse triangular systems when an approximation is suitable. This approach will not work for all problems, but can be successful for sparse triangular matrices arising from incomplete factorizations, where an approximate solution is acceptable. We demonstrate the performance gains that this approach can have on GPUs in the context of solving sparse linear systems with a preconditioned Krylov subspace method. We also illustrate the effect of using asynchronous iterations.

1 Introduction

Solves with sparse triangular matrices are difficult to parallelize efficiently, due to the often irregular structure of sparse matrices and the sequential nature of forward and backward substitution. The most common way to parallelize sparse triangular solves is to use a "level scheduling" technique [22]. A "level" consists of the unknowns that can be computed in parallel, given the dependency graph implied by the sparse matrix. The levels are processed in sequence until all the unknowns are computed. Depending on the sparse matrix, there may be a very large number of levels or not enough work within a level to efficiently utilize highly parallel architectures such as graphics processing units (GPUs).

In this paper, we investigate the approach of using an iterative method to solve sparse triangular systems. It is unconventional to apply iterative methods to triangular systems because such systems can be solved directly. However, due to high efficiency sparse-matrix vector product codes that have been vigorously developed in recent years, including on GPUs, iterations with sparse triangular matrices can be very fast compared to forward and backward substitution. In this paper, we use the Jacobi iterative method, although nonstationary methods and polynomial methods can also be used. Because triangular matrices are non-normal, the Jacobi method may diverge and cause overflow before converging, depending on the degree of non-normality of the matrix. However, for many types of sparse triangular matrices, such as the triangular parts of matrices

J.L. Träff et al. (Eds.): Euro-Par 2015, LNCS 9233, pp. 650–661, 2015.
DOI: 10.1007/978-3-662-48096-0_50

from discretizations of partial differential equations, and from incomplete factorizations of these matrices, the triangular matrices have a degree of diagonal dominance that can avoid divergence of the Jacobi iterations. Thus, although Jacobi iterations will not work for all matrices, there are large, useful classes of matrices for which Jacobi iterations can be a viable approach for solving sparse triangular systems.

The iterative approach taken here is particularly applicable and competitive when only an approximate solution is sought, meaning, only a small number of Jacobi iterations are necessary. This is the situation when the triangular solves are used in preconditioned Krylov subspace methods for solving linear systems. Here, the triangular matrices themselves, such as from incomplete factorizations, are only approximations and approximate solves are acceptable when applying the preconditioner. By using approximate solves, the total number of iterations of the Krylov subspace method may be larger than when exact solves are used, but the total execution time may be much smaller.

We investigate the use of Jacobi iterations (also called sweeps in this paper) and a "block-asynchronous" variant to apply an incomplete LU (ILU) factorization preconditioner. The asynchronous variant does not synchronize the updates of variables within each sweep and may have improved convergence rate and execution time. When a fixed number of synchronous Jacobi sweeps are used, the operator is fixed, and standard Krylov subspace methods may be used. For the asynchronous variant, the operator is not fixed and therefore we use a flexible method, in particular, flexible GMRES (F-GMRES) [21] which we have implemented in the MAGMA [13] library for GPUs.

The acceleration of sparse triangular solves is the subject of much current research, e.g., [19], but almost all this research is based on the level scheduling idea [2,12,23]. Efficient implementations on state-of-the-art hardware still pose a challenge [14,15,26]. Another approach to parallelizing sparse triangular solves is to use partitioned inverses [1,20]. Here, a triangular matrix is written as a product of sparse triangular factors; each triangular solve is then a sequence of sparse matrix vector multiplications. The use of a sparse approximate inverse for a triangular matrix has been considered in [10,24], as well as the idea of approximating the inverse ILU factors via a truncated Neumann series [24,25]. The latter is similar to the idea of using Jacobi sweeps presented in this paper. The use of Jacobi sweeps for for sparse triangular solves was recommended in [6] for the Intel MIC architecture. Asynchronous iterations for these sweeps were not considered. The potential of replacing synchronous Jacobi with block-asynchronous Jacobi for more efficient use of the GPU hardware was investigated in [3] and applied to smoothers for geometric multigrid methods in [4].

This paper is organized as follows. Section 2 first provides some background and Sect. 3 gives details about the actual implementations for the methods we use for the experimental part (Sect. 4) in this paper. Section 4.1 describes our test environment, in Sect. 4.2 we compare the convergence of classical Jacobi and a block-asynchronous version when solving sparse triangular systems. For the latter, we investigate the effect of scheduling the GPU thread blocks consistently with the data dependency order in the triangular factors. We also compare the execution time with level-scheduling triangular solves. In Sect. 4.3 we investigate the impact of approximate triangular solves when used in preconditioned F-GMRES(50). We conclude in Sect. 5.

2 Background

2.1 Jacobi Method and Asynchronous Iteration

Classical relaxation methods like Jacobi or Gauss-Seidel are defined using a specific update order of the vector components and imply synchronization between the distinct iterations. The number of components that can be computed in parallel in an iteration depends on whether the update of a component uses only information from the previous iteration (Jacobi type) or also information from the current iteration (Gauss-Seidel type). Using newer information generally results in faster convergence. This however comes at the price of reduced parallelism: Gauss-Seidel is inherently sequential and requires a strict update order; for Jacobi, all components are updated simultaneously within one iteration. Asynchronous relaxation methods do not obey any update order. Instead they iterate the components in a nondeterministic fashion, always using the newest available values of the other components. The implied fine-grained parallelism and the lack of synchronization makes asynchronous methods attractive for GPUs, which themselves operate in an asynchronous-like fashion. At the same time, asynchronous iteration methods require the target matrix to have stronger properties to ensure convergence. For the asynchronous relaxation suitable for linear systems, a sufficient condition for convergence is given if the spectral radius of the positive iteration matrix, $\rho(|M|)$, is smaller than unity [11]. If this convergence condition is fulfilled, a block-asynchronous Jacobi iteration, where subsets of components are iterated in synchronous fashion and asynchronous updates are used in-between the subsets, also converges [5]. The Jacobi iteration for solving $Ax = b$ can be written as

$$\begin{aligned} x^{k+1} &= D^{-1}\left(b - (A - D)x^k\right) \\ x^{k+1} &= D^{-1}b + Mx^k \end{aligned} \tag{1}$$

where D the diagonal part of A [5]. For the triangular systems that arise in the context of incomplete factorization preconditioning, we denote the iteration matrices as M_L and M_U for the lower and upper triangular, respectively. Let D_L and D_U be the diagonal parts of the triangular factors L and U, and let I be the identity matrix. For the diagonal of L being all ones,

$$M_L = D_L^{-1}(D_L - L) = I - L, \qquad M_U = D_U^{-1}(D_U - U) = I - D_U^{-1}U. \tag{2}$$

Hence, M_L is strictly lower triangular and M_U is strictly upper triangular, which implies that the spectral radius of both iteration matrices is zero. Therefore, the asynchronous method converges in the asymptotic sense for any triangular system [11].

2.2 Incomplete LU Preconditioning

An ILU factorization is the approximate factorization of a nonsingular sparse matrix A into the product of a sparse lower triangular matrix L and a sparse upper triangular matrix U, $A \approx LU$, where nonzeros or fill-in are only permitted in specified locations. The basic algorithm, called ILU(0), approximates the LU factorization by allowing only nonzero elements in L and U that are nonzero in A. In this case, the sparsity pattern of L and U matches the sparsity pattern of the system matrix A. The ILU factorization can also be computed efficiently in parallel on GPUs using a fine-grained parallel iterative algorithm [6,7].

3 Block-Asynchronous Jacobi on GPUs

If we allow Jacobi to use newer information in the component updates, the resulting asynchronous iteration can be realized in a single kernel that overwrites the iteration input vector with the updated values. The algorithm may be considered as block-asynchronous Jacobi as components handled by the same GPU thread block are updated simultaneously in Jacobi fashion, but the distinct thread blocks are executed asynchronously without enforcing a certain update order. Using newer information from remote components has the potential of improving the convergence, but carries the danger of degraded convergence if some components are updated several times in a row without using newer information about the other components [5].

The order in which the components are updated depends on the scheduling of the GPU thread blocks. GPUs use the concept of thread blocks to apply a kernel operation to data, and typically not all data is processed at the same time, but some thread blocks are scheduled before others [16]. The components handled by one GPU thread block are updated in parallel using the newest available information for the other components. Unfortunately, GPUs generally do not allow insight or modifications to the thread block execution order. However, backward-engineering experiments reveal that the thread blocks are usually scheduled in consecutive increasing order. With Gauss-Seidel converging usually faster than Jacobi, this motivates us to update the components in dependency order. For triangular matrices with a small bandwidth, i.e. the triangular factors arising from the RCM-reordered systems, this effect may be small. For matrix entries with a distance to the diagonal larger than the thread block size, updating in dependency order is equivalent to an Gauss-Seidel update, which would be equivalent to an exact substitution for this matrix component. For the lower triangular solve, updating the components in dependency order is equivalent to scheduling the thread blocks in consecutive increasing order. For the upper triangular solve, this scheduling order is against the dependency order, and faster convergence should be achieved by reversing the scheduling order. We investigate the effect of the thread block scheduling order in Sect. 4.2.

For classical (synchronous) Jacobi, the thread block scheduling has no impact, as no new information from the current iterate is used. This however implies, that the algorithm's implementation can not be realized in a single kernel overwriting the iteration input vector with its output (neglecting the case of the hardware parallelism being larger than the iteration vector length). Either separate input/output vectors have to be used, or the Jacobi is realized in two kernels where the first computes the sparse matrix vector product and the second performs the update.

4 Experimental Results

4.1 Test Environment

The experimental results were obtained using a Tesla K40 GPU (Kepler microarchitecture) with a theoretical peak performance of 1,682 GFlop/s (double precision). The 12 GB of GPU main memory, accessed at a theoretical bandwidth of 288 GB/s, was sufficiently large to hold all the matrices and all the vectors needed in the iteration process. Although all operations are handled by the accelerator, we mention for completeness that the host was being an Intel Xeon E5 processor (Sandy Bridge). The implementation of all GPU kernels is realized in CUDA [16], version 7.0 [18], using a thread block size of 128, NVIDIA's sparse matrix vector product was taken from the cuSPARSE library version 7.0 [17]. Double precision computations were used. To account for the non-deterministic properties of the asynchronous methods, the reported results are averaged over 50 runs.

Nonsymmetric test matrices were selected from the University of Florida sparse matrix collection (UFMC) [8], and are listed in Table 1. We also included a test matrix arising from a finite difference discretization of the Laplace operator in 3D with Dirichlet boundary conditions. A 27-point stencil was used on a $64 \times 64 \times 64$ mesh. Although this latter matrix is symmetric, we treat it as nonsymmetric in our experimental tests. The sparsity plots for all test matrices are given in Fig. 1.

Reverse Cuthill-McKee (RCM) ordering is well-known to reduce the matrix bandwidth and can produce more accurate incomplete factorization preconditioners [9]. Except for the DC test problem where RCM reordering fails to reduce the bandwidth, we consider all test matrices in RCM ordering (all matrices have symmetric structure). Note that we do not use multicolor orderings as these typically degrade the approximation properties of ILU preconditioners [9] although these orderings can enhance the parallelism for level scheduling.

Table 1. Test matrices.

	Name	Abbrev.	Description	Nonzeros n_z	Size n
UFMC	CHIPCOOL0	CHP	Convective thermal flow (FEM)	281,150	20,082
	DC1	DC	Circuit simulation matrix	766,396	116,835
	STOMACH	STO	3D electro-physical duodenum model	3,021,648	213,360
	VENKAT01	VEN	Unstructured 2D Euler solver (FEM)	1,717,792	62,424
	LAPLACE3D	LAP	3D Laplace problem (27-pt stencil)	6,859,000	262,144

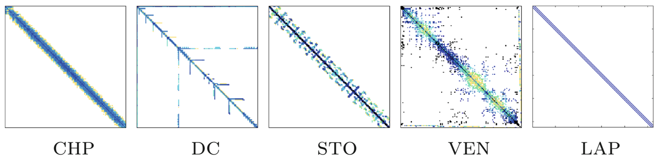

Fig. 1. Sparsity plots of test matrices listed in Table 1.

4.2 Sparse Triangular Solves

In this section, we report experimental results on convergence and performance when solving sparse triangular systems with relaxation methods. In these systems, the right-hand side is the vector of all ones, and the initial guess is the zero vector. Figures 2 and 3 show results for ILU(0) factors from two very different test problems, LAP and DC, respectively. In each figure, the top set of

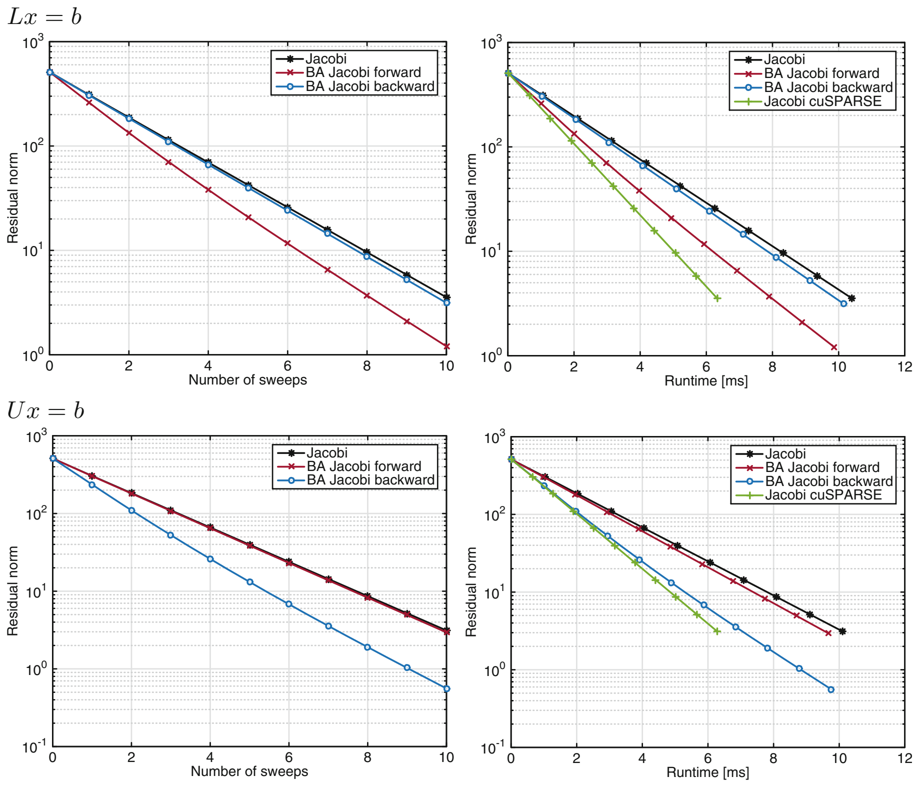

Fig. 2. For the lower and upper triangular ILU(0) factors of the LAP problem, convergence (left) and runtime (right) of the synchronous Jacobi and the block-asynchronous Jacobi (averaged results).

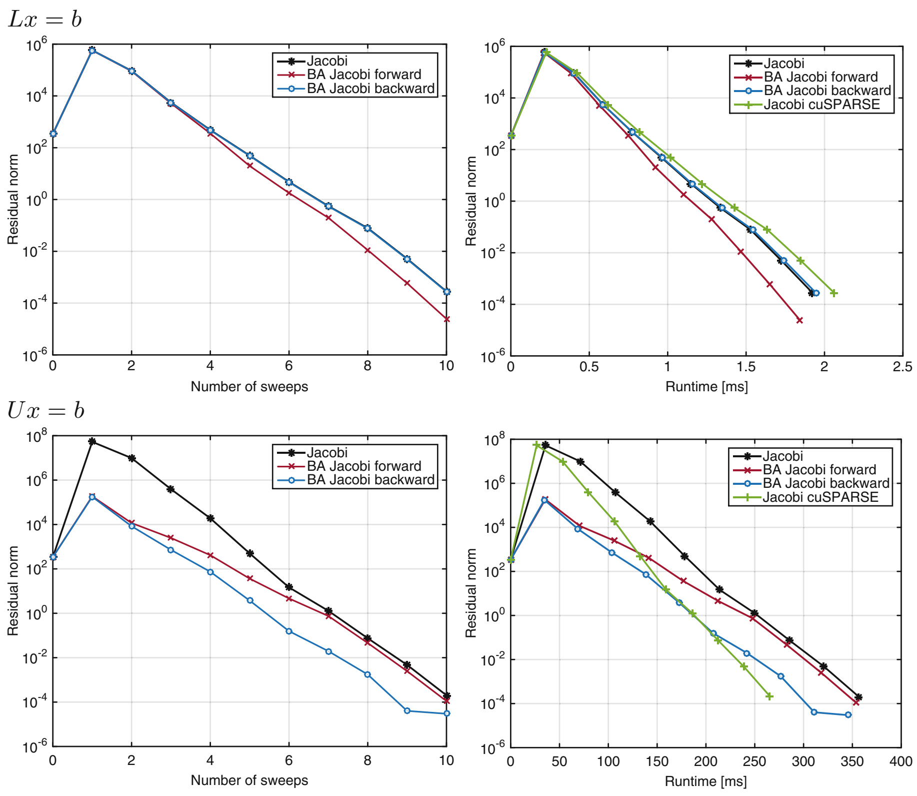

Fig. 3. For the lower and upper triangular ILU(0) factors of the DC problem, convergence (left) and runtime (right) of the synchronous Jacobi and the block-asynchronous Jacobi (averaged results).

graphs show results for lower triangular solves, and the bottom set of graphs show results for upper triangular solves.

The results show that the block-asynchronous methods converge faster than the classical Jacobi methods. In the lower triangular case, forward thread block ordering gives faster convergence than backward thread block ordering, as predicted in Sect. 3. The convergence of backward thread block ordering is very similar to that of classical Jacobi, as the method tends not to use newly computed information within an iteration. The opposite of the above statements is true for the upper triangular case.

The timing results follow the same trends as the convergence results. We note that for the DC problem, the timings for the upper triangular solves are much higher than the timings for the lower triangular solves. This will be explained at the end of this subsection.

The graphs also show results for a Jacobi implementation based on the sparse matrix vector product from NVIDIA's cuSPARSE library [17]. Naturally, the

Table 2. Runtime comparison [ms] between the exact triangular solve using the cuSPARSE level-scheduling implementation and one Jacobi sweep.

Matrix	Factor	Exact triangular solve	Single Jacobi sweep
CHP	L	7.84	0.10
	U	7.07	0.10
DC	L	0.62	0.23
	U	4.65	26.57
STO	L	21.61	0.40
	U	24.16	0.37
VEN	L	17.49	0.23
	U	14.81	0.23
LAP	L	12.13	0.66
	U	11.57	0.65

optimization level of this routine is significantly higher than of our CUDA based implementations. In the end, the cuSPARSE based Jacobi is the overall winner in terms of runtime. However, from comparing the results of synchronous and block-asynchronous Jacobi, it can be deduced that applying the same level of optimization to the kernel for block-asynchronous iteration would make it superior also to the cuSPARSE based Jacobi. In the remainder of the paper we use the cuSPARSE based Jacobi implementation. There is no "asynchronous" sparse matrix vector product in cuSPARSE (which would give "approximate" and nondeterministic results) that we could use to implement a more efficient block-asynchronous Jacobi kernel.

Table 2 compares the runtime for exact sparse triangular solves from the NVIDIA cuSPARSE library to the runtime of one single Jacobi sweep. The data reveals that a Jacobi sweep typically costs a fraction of the total time for an exact sparse triangular solve using level scheduling (although multiple sweeps will generally be needed for an approximate solve). Only for the test case DC, which comes from circuit simulation modeling, one Jacobi sweep on the upper triangular system is more expensive than the level-scheduling exact solve. The reason for this is the structure of this matrix: very unbalanced lengths of rows in the upper triangular part of this matrix (see Fig. 1 for the DC matrix, where some rows have many more nonzeros than others) causes load imbalance in the GPU kernels. Performance could be improved by using a load balanced sparse matrix vector product kernel.

4.3 ILU-Preconditioned FGMRES

Figures 4 and 5 show the impact of replacing the exact triangular solves by approximate triangular solves in an ILU(0) preconditioned F-GMRES(50) solver. The left side of these figures relates the number of relaxation sweeps in the

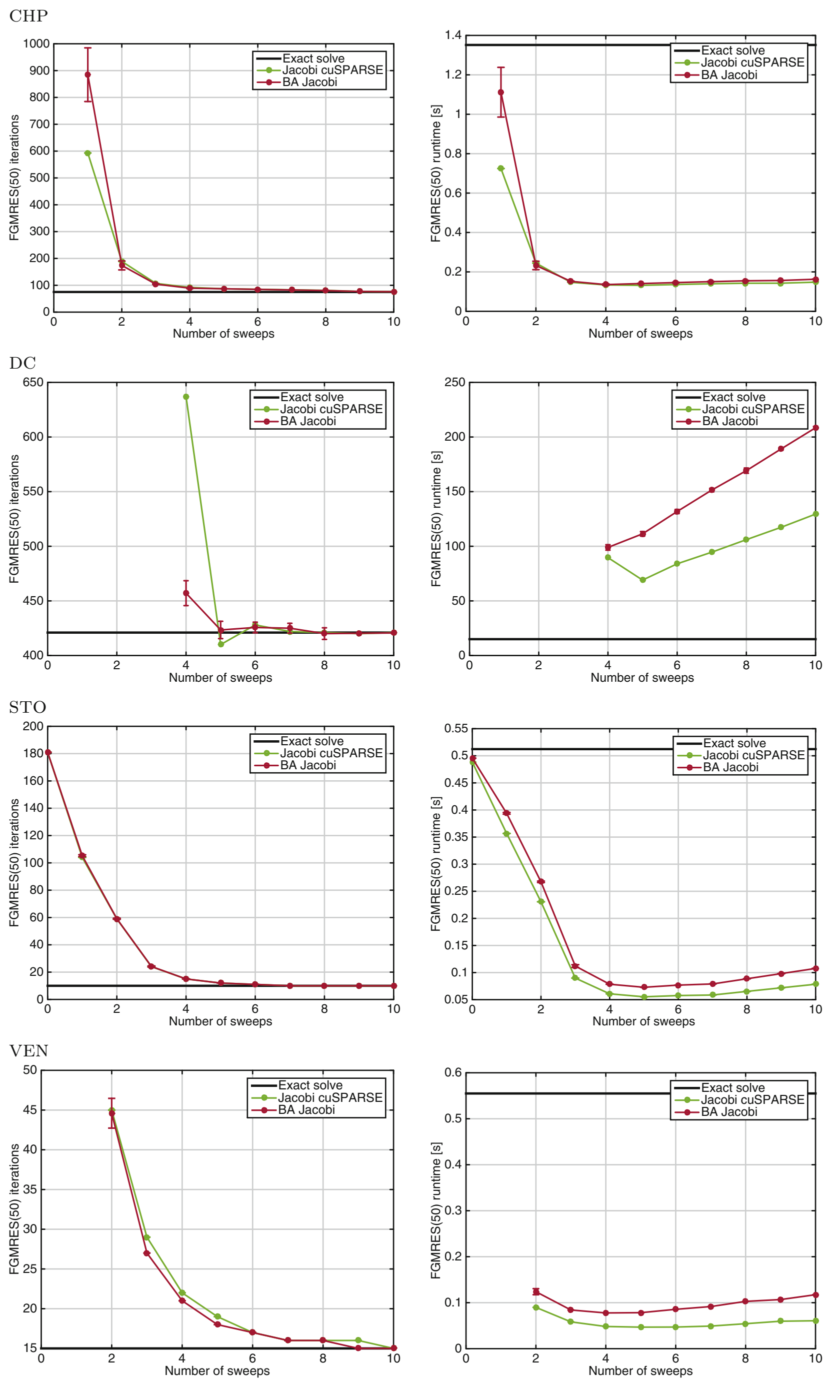

Fig. 4. F-GMRES(50) convergence (left) and runtime (right) when using either exact or approximate triangular solves for the test matrices from UFMC.

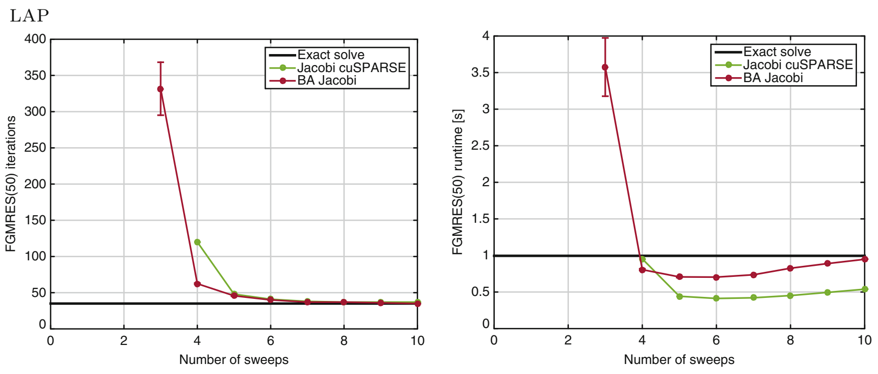

Fig. 5. F-GMRES(50) convergence (left) and runtime (right) when using either exact or approximate triangular solves for the LAP test case.

approximate triangular solve to the F-GMRES(50) iteration count. According to the convergence results on the triangular solves in the previous section, the block-asynchronous Jacobi algorithm schedules the thread blocks in dependency order. This is forward thread block scheduling when solving the lower triangular systems, and backward thread block scheduling when solving the upper triangular systems. In many cases, the faster convergence of the block-asynchronous Jacobi accounting for the dependency order compared to the synchronous Jacobi is reflected in the top level solver convergence: e.g., for CHP, VEN and LAP, the left-hand side plots show that on average, fewer F-GMRES(50) iterations are required for block-asynchronous Jacobi than for classical Jacobi. The error bars for block-asynchronous Jacobi indicate one standard deviation above and below the mean. They reveal that especially when using only few sweeps of block-asynchronous Jacobi, significant variation in the solver iterations may be expected. For systems with most entries close to the diagonal, the standard deviation is very small, and the iteration counts are almost identical to those using synchronous Jacobi. In general, few sweeps of the approximate triangular solve are sufficient to get the same F-GMRES(50) iteration count like when using exact triangular solves.

The right-hand side of Fig. 4 (respectively Fig. 5 for the LAP problem) relates the F-GMRES(50) convergence with respect to the runtime. Applying few sweeps of the relaxation method is usually less expensive than a level-scheduling exact solve, and can reduce the top-level solver execution time. In particular, the faster preconditioner application can compensate for a few additional iterations. Except for the DC problem (Fig. 4), where the sparse matrix vector product suffers from the unbalanced nonzero distribution, all problems benefit from replacing the level scheduling triangular solve by an approximate solve in the preconditioner application. We noticed that synchronous Jacobi usually requires a few additional F-GMRES(50) iterations. In terms of performance, synchronous Jacobi still beats block-asynchronous Jacobi. This is due to the performance of the cuSPARSE sparse matrix vector kernel that the synchronous Jacobi is based on. As previously mentioned, the block-asynchronous Jacobi would likely outperform the synchronous counterpart if it were optimized the same way.

Separate experiments on a consumer card of NVIDIA's Kepler architecture (not shown here) revealed that block-asynchronous Jacobi becomes even more attractive when there is less hardware parallelism. This is due to the fact that fewer GPU thread blocks can be scheduled simultaneously, resulting in a higher ratio of Gauss-Seidel-to-Jacobi-type of updates, which improves convergence.

5 Conclusions

We investigated the potential of approximate triangular solves for an incomplete LU factorization preconditioner on GPU accelerators, replacing the level-scheduled exact forward and backward substitutions with classical and block-asynchronous Jacobi iterations allowing for fine-grained parallelism. We analyzed the trade-off between convergence penalty caused by lower preconditioning accuracy and enhanced parallelism for several test matrices. We have shown that few sweeps of an iterative method are often sufficient to provide the same preconditioner quality as the top-level solver. Even if additional iterations are required by the approximate triangular solve, they are in many cases compensated by faster preconditioner application. Future research will focus on porting the approximate triangular solve to other hardware architectures, and investigating the potential of faster information propagation by adding local Jacobi sweeps for cached values for components handled by the same thread block, and using overlapping iteration blocks.

Acknowledgments. This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, Applied Mathematics program under Award Numbers DE-SC-0012538 and DE-SC-0010042. Support from NVIDIA is also gratefully acknowledged.

References

1. Alvarado, F.L., Schreiber, R.: Optimal parallel solution of sparse triangular systems. SIAM J. Sci. Comput. **14**, 446–460 (1993)
2. Anderson, E.C., Saad, Y.: Solving sparse triangular systems on parallel computers. Intl. J. High Speed Comput. **1**, 73–96 (1989)
3. Anzt, H., Tomov, S., Dongarra, J., Heuveline, V.: A block-asynchronous relaxation method for graphics processing units. J. Parallel Distrib. Comput. **73**(12), 1613–1626 (2013)
4. Anzt, H., Tomov, S., Gates, M., Dongarra, J., Heuveline, V.: Block-asynchronous Multigrid Smoothers for GPU-accelerated Systems. In: ICCS. Procedia Computer Science, vol. 9, pp. 7–16. Elsevier (2012)
5. Anzt, H.: Asynchronous and Multiprecision Linear Solvers - Scalable and Fault-Tolerant Numerics for Energy Efficient High Performance Computing. Ph.D. thesis, Karlsruhe Institute of Technology, Institute for Applied and Numerical Mathematics, Nov 2012

6. Chow, E., Patel, A.: Fine-grained parallel incomplete LU factorization. SIAM J. Sci. Comput. **37**, C169–C193 (2015)
7. Chow, E., Anzt, H., Dongarra, J.: Asynchronous iterative algorithm for computing incomplete factorizations on GPUs. In: Kunkel, J.M., Ludwig, T. (eds.) ISC High Performance 2015. LNCS, vol. 9137, pp. 1–16. Springer, Heidelberg (2015)
8. Davis, T.A.: University of Florida Sparse Matrix Collection. na-digest 92 (1994)
9. Duff, I.S., Meurant, G.A.: The effect of ordering on preconditioned conjugate gradients. BIT **29**(4), 635–657 (1989)
10. Duin, A.C.N.V.: Scalable parallel preconditioning with the sparse approximate inverse of triangular matrices. SIAM J. Matrix Anal. Appl. **20**, 987–1006 (1996)
11. Frommer, A., Szyld, D.B.: On asynchronous iterations. J. Comput. Appl. Math. **123**, 201–216 (2000)
12. Hammond, S.W., Schreiber, R.: Efficient ICCG on a shared memory multiprocessor. Intl. J. High Speed Comput. **4**, 1–21 (1992)
13. Innovative Computing Lab: Software distribution of MAGMA version 1.6 (2015). http://icl.cs.utk.edu/magma/
14. Mayer, J.: Parallel algorithms for solving linear systems with sparse triangular matrices. Computing **86**(4), 291–312 (2009)
15. Naumov, M.: Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. Technical report NVR-2011-001, NVIDIA (2011)
16. NVIDIA Corporation: NVIDIA CUDA Compute Unified Device Architecture Programming Guide, 2.3.1 edn., August 2009
17. NVIDIA Corporation: CUSPARSE LIBRARY V7.0, March 2015
18. NVIDIA Corporation: NVIDIA CUDA TOOLKIT V7.0, March 2015
19. Park, J., Smelyanskiy, M., Sundaram, N., Dubey, P.: Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In: Kunkel, J.M., Ludwig, T., Meuer, H.W. (eds.) ISC 2014. LNCS, vol. 8488, pp. 124–140. Springer, Heidelberg (2014)
20. Pothen, A., Alvarado, F.: A fast reordering algorithm for parallel sparse triangular solution. SIAM J. Sci. Statis. Comput. **13**(2), 645–653 (1992)
21. Saad, Y.: A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci. Comput. **14**(2), 461–469 (1993)
22. Saad, Y.: Iterative Methods for Sparse Linear Systems. SIAM, Philadelphia (2003)
23. Saltz, J.H.: Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM J. Sci. Stat. Comput. **11**, 123–144 (1990)
24. Tuma, M., Benzi, M.: A comparative study of sparse approximate inverse preconditioners. Appl. Numer. Math. **30**, 305–340 (1998)
25. van der Vorst, H.: A vectorizable variant of some ICCG methods. SIAM J. Sci. Statis. Comput. **3**(3), 350–356 (1982)
26. Wolf, M.M., Heroux, M.A., Boman, E.G.: Factors impacting performance of multithreaded sparse triangular solve. In: Daydé, M., Lopes, J.C., Marques, O., Palma, J.M.L.M. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 32–44. Springer, Heidelberg (2011)