Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs

Hartwig Anzt
Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee, USA
hanzt@icl.utk.edu

Jack Dongarra
Innovative Computing Lab, University of Tennessee, Knoxville, Tennessee, USA; Oak Ridge National Laboratory, USA; School of Computer Science, University of Manchester, United Kingdom
dongarra@icl.utk.edu

Goran Flegar
Enrique S. Quintana-Ortí
Depto. Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain
flegar@uji.es, quintana@uji.es

Abstract

In this paper, we design and evaluate a routine for the efficient generation of block-Jacobi preconditioners on graphics processing units (GPUs). Concretely, to exploit the architecture of the graphics accelerator, we develop a batched Gauss-Jordan elimination CUDA kernel for matrix inversion that embeds an implicit pivoting technique and handles the entire inversion process in the GPU registers. In addition, we integrate extraction and insertion CUDA kernels to rapidly set up the block-Jacobi preconditioner.

Our experiments compare the performance of our implementation against a sequence of batched routines from the MAGMA library realizing the inversion via the LU factorization with partial pivoting. Furthermore, we evaluate the costs of different strategies for the block-Jacobi extraction and insertion steps, using a variety of sparse matrices from the SuiteSparse matrix collection. Finally, we assess the efficiency of the complete block-Jacobi preconditioner generation in the context of an iterative solver applied to a set of computational science problems, and quantify its benefits over a scalar Jacobi preconditioner.

Categories and Subject Descriptors G.4 [*Mathematical Software*]: Efficiency; C.4 [*Performance of systems*]: Performance and energy efficiency; C.1.3 [*Computer Systems Organization*]: Other Architecture Styles—heterogeneous (hybrid) systems

Keywords Sparse linear systems, iterative methods, block-Jacobi preconditioner, matrix inversion, Gauss-Jordan elimination, graphics processing units (GPUs)

1. Introduction

Preconditioning is a crucial task for the efficient solution of large-scale sparse linear systems via iterative methods [16]. The challenge is to find a preconditioner for the linear system that accelerates the convergence of the iterative scheme. In practice, the preconditioned iteration is attractive if the improvement of the convergence rate compensates the additional work of 1) calculating the preconditioner; and 2) applying the preconditioner during the iteration process. In the context of high performance computing, the efficiency of the preconditioner depends on how well these two building blocks, preconditioner calculation and application, scale on parallel architectures.

Incomplete LU (ILU) factorization techniques comprise some of the most effective preconditioners [16]. These methods generate the preconditioner as an incomplete factorization of the system matrix on some nonzero pattern. The preconditioner application in the distinct iteration steps requires solving sparse triangular systems for the incomplete factors in order to implicitly transform the original problem into the preconditioned one. Unfortunately, the convergence acceleration due to ILU preconditioning comes at the price of introducing the hard-to-parallelize sparse triangular solves. As these triangular kernels can easily become a bottleneck on parallel architectures, much effort has been spent on developing efficient strategies that replace forward and backward substitutions with approximations that provide better scalability; see [1, 3, 7, 8] and references therein.

Compared with ILU, preconditioners based on Jacobi (diagonal scaling) and block-Jacobi typically render lower acceleration rates on the convergence of the iterative solver [16]. In contrast, the application of a Jacobi-type preconditioner is an inherently-parallel operation, which turns these strategies highly appealing for massively-parallel systems. In particular, on many-core data-parallel architectures, such as graphics processing units (GPUs), Jacobi-type preconditioners introduce a negligible overhead. Furthermore, as the sparse matrix-vector product underlying the Jacobi-type preconditioners is a central component for many sparse linear algebra methods, hardware-optimized versions are typically available.

From a practical point of view, the preconditioner setup for a Jacobi scheme requires extracting and inverting the main diagonal of the coefficient matrix for the linear system. For problems that inherently carry a block structure, such as for example higher-order finite element discretizations, block-Jacobi preconditioners typically offer higher convergence benefits. However, a block-Jacobi scheme is more expensive as it involves extracting the diagonal blocks from the coefficient matrix, which is typically stored in a sparse data structure; and either inverting these diagonal blocks for generating a block-Jacobi matrix, or solving a set of small linear systems in every preconditioner application.

In this paper we propose a batched routine that generates a block-Jacobi preconditioner via the explicit inversion of a collection of small dense linear systems, especially tailored for graphics accelerators. For this purpose, we design a batched routine, based on Gauss-Jordan elimination (GJE) [11], that benefits from an im-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PMAM'17 February 4–8, 2017, Austin, Texas, USA
Copyright © 2017 ACM 978-1-4503-4883-6/17/02...$15.00
DOI: http://dx.doi.org/10.1145/10.1145/3026937.3026940

plicit pivoting strategy and handles the inversion process in the GPU registers. In addition, we combine the batched Gauss-Jordan elimination (BGJE) routine with the efficient extraction of the appropriate matrix entries from the sparse data structures. For this extraction step, we propose different strategies that trade-off pressure on the memory bandwidth, coalescent memory access, and additional use of shared memory. Our experimental results reveal that there exist no overall winner strategy, but the performance strongly depends on hardware technology, Jacobi block size, and matrix properties. To illustrate this point, we provide a detailed analysis on the overhead that the data extraction+insertion adds to BGJE, and a comprehensive performance analysis comparing BGJE with other batched routines designed for the inversion of small dense matrices.

2. Background and Related Work

2.1 Block-Jacobi preconditioning

Consider the linear system $Ax = b$, where the coefficient matrix $A \in \mathbb{R}^{n\times n}$, the right-hand side vector $b \in \mathbb{R}^n$, and the sought-after solution $x \in \mathbb{R}^n$. The Jacobi method splits the coefficient matrix as $A = L + D + U$, where D denotes the diagonal of A (or block diagonal for all block-Jacobi methods), and L/U respectively contain the entries of A below/above those in D. For a starting solution guess $x^{\{0\}}$, a Jacobi-type iteration based on this splitting can then be formulated as:

$$\begin{aligned} x^{\{k\}} &:= D^{-1}\left(b-(A-D)x^{\{k-1\}}\right) \\ &= D^{-1}b + Mx^{\{k-1\}}, \qquad k = 1,2,\dots \end{aligned} \tag{1}$$

The convergence of a block-Jacobi iteration is guaranteed if the spectral radius of the iteration matrix M fulfills [16]

$$\rho(M) = \rho\left(I - D^{-1}A\right) < 1.$$

This is fulfilled for diagonally-dominant systems [16]. When used as a preconditioner, the relaxation is typically reduced to the (block) diagonal scaling of the right-hand side vector:

$$x := D^{-1}b. \tag{2}$$

Restricting the Jacobi relaxation to a (block) diagonal scaling ignores the term $x - D^{-1}Ax$ in (1). Although, in general, this term is nonzero, (block) diagonal scaling often succeeds in enhancing the convergence of the iteration.

When used within an iterative framework, the application of a block-Jacobi preconditioner $D = \text{diag}(D_1, D_2, \dots, D_N)$ either requires the solution of the block diagonal linear system (2) or (assuming the block-inverse $\hat{D} = D^{-1} = \text{diag}(D_1^{-1}, D_2^{-1}, \dots, D_N^{-1})$ has been explicitly pre-computed) a block-diagonal scaling in terms of a matrix-vector multiplication. Typically, pre-computing the block-inverse matrix $\hat{D}$ explicitly in the preconditioner setup is more attractive as it allows for faster preconditioner application in the iterative solver. However, when dealing with large blocks and sparse data structures, the inversion of matrix D can become a bottleneck. To tackle this, we handle each block D_i separately, and use the GJE to generate their individual inverses.

2.2 GJE for matrix inversion

The conventional procedure to invert a matrix (block) D_i consists of four steps that commence with the computation of the LU factorization (with partial pivoting): $P_iD_i = L_iU_i$, where L_i is lower unit triangular, U_i is upper triangular, and P_i is a permutation, all three of the same dimension as the original matrix [9]. This is followed by the inversion of the upper triangular factor $\hat{U}_i := U_i^{-1}$; the lower triangular solve $\hat{D}_i := \hat{U}_iL_i^{-1}$; and the back-transform of the permutation $D_i^{-1} := \hat{D}_iP_i$.

```
 1 % Input  : m x m nonsingular matrix block Di.
 2 % Output : Matrix block Di overwritten by its inverse
 3 p = [1:m];
 4 for k = 1 : m
 5    % pivoting
 6    [abs_ipiv, ipiv]        = max(abs(Di(k:m,k)));
 7    ipiv                    = ipiv+k-1;
 8    [Di(k,:), Di(ipiv,:)] = swap(Di(ipiv,:), Di(k,:));
 9    [p(k),   p(ipiv)]      = swap(p(ipiv), p(k));
10
11    % Jordan transformation
12    d        =  Di(k,k);
13    Di(:,k) =-[Di(1:k-1,k); 0; Di(k+1:m,k)] / d;% SCAL
14    Di       =   Di + Di(:,k) * Di(k,:);          % GER
15    Di(k,:) = [Di(k,1:k-1), 1, Di(k,k+1:m)] / d;% SCAL
16 end
17 % Undo permutations
18 Di(:,p) = Di;
```

Figure 1. Simplified loop-body of the basic GJE implementation in Matlab notation.

The inversion of large dense matrices via GJE has been recently revisited as an efficient alternative for current parallel systems, including clusters and GPU accelerators [6, 15]. In essence, matrix inversion via GJE combined with partial pivoting is as stable as the LU-based approach, but avoids the workload unbalance due to the operation with triangular factors in the four-stage procedure. Furthermore, as we will describe in Section 2.3, matrix inversion based on GJE allows an implicit permutation of the matrix.

From the algorithmic point of view, GJE for matrix inversion consists of a loop that applies two vector scalings (SCAL) and a general rank-1 update of the matrix (GER) at each iteration of the algorithm; see Figure 1. The sequence of permutations produced by GJE is the same as in the LU factorization with partial pivoting, and GJE can be viewed as a reorganization of the four-stage LU-based inversion approach [15]. While there exist blocked formulations of GJE that introduce Level-3 BLAS to increase the re-utilization of data in the cache for large-scale dense matrices, the algorithm based on Level-2 BLAS operations in Figure 1 achieves good performance for the small matrices we address.

2.3 GJE with implicit pivoting

Performing the pivoting step as described in Figure 1 requires additional data movements to exchange the contents of the `k`-th row with those of the selected `ipiv`-th row at each step. By inspecting the operations involved in the Jordan transformation though, we observe that the transformation applied to each row is only affected by the values in that particular row and the selected `ipiv`-th row. Hence, the actual order of the rows in the matrix is not important during the inversion process. Thus, GJE can be implemented without actually swapping the rows during the inversion process. Instead, all pivoting steps can be accumulated, and be realized afterwards. When implementing this alternative approach, the pivoting step requires some minor modifications as the pivot candidates are no longer those elements belonging to rows `k:n`. Instead the algorithm needs to keep track of which rows were not yet used as pivots and select the next pivot among them. In addition, the accumulated row permutations have to be applied, together with the column permutations, after the inversion process is completed. Avoiding these data movements is especially appealing for massively parallel architectures, such as GPUs.

Since using implicit pivoting does not change the execution order of the operations but only differs in the data's memory location, the numerical stability of the GJE algorithm is not affected by the implicit pivoting strategy.

2.4 Related work on batched GPU routines

The term "batched" refers to routines that repeatedly apply the same operation to a large collection of independent data entities. These entities are typically small, and the absence of dependencies makes the problem embarrassingly parallel. On parallel systems, applying the operation to a single data entity may not fully utilize the hardware as scheduling one data entity after another may leave computational resources unused. A batched implementation applies the operation to several/all data entities simultaneously, and hence allows for more significant use of the parallel hardware. On streaming processors, such as GPUs, an additional advantage comes from the reduced kernel launch overhead from a single batched kernel call, versus making multiple calls, one for each linear system. Furthermore, if the data entities are stored consecutively in the GPU main memory, a batched routine can make more efficient use of the memory access as, e.g. on NVIDIA GPUs, each memory transaction can read or write a few bytes of contiguous memory. Examples of the superiority of batched implementations over baseline implementations often focus on BLAS kernels such as batched matrix-matrix multiplication, matrix factorizations for linear systems, and triangular systems solves [10, 13]. The use of batched routines for efficient preconditioner generation has recently also been studied in the context of using approximate triangular solves for incomplete factorization preconditioning [2].

3. Design of CUDA Kernels

In order to generate a block-Jacobi preconditioner that operates with the explicit inverses of the diagonal blocks D_i, these submatrices have to be extracted from the coefficient matrix A, be inverted, and written back into the preconditioner matrix $\hat{D}$. As the coefficient matrix as well as the preconditioner are typically stored using sparse data structures, neither the extraction nor the insertion are straight-forward. Therefore, an algorithm that generates a block-Jacobi preconditioner may handle these steps separately from the inversion of the diagonal blocks, with the latter computation realized via the previously reviewed GJE.

Figure 2 illustrates the organization into three steps of the algorithm that generates the block-Jacobi preconditioner. Here we note that the inverse of a sparse matrix is in general dense, and in order to attain an efficient processing in terms of a batched routine, we convert the diagonal blocks to dense format. In the remainder of this section we provide details about these steps, and their efficient realization on GPUs. For completeness we mention that identifying the diagonal blocks via supervariable agglomeration and/or graph partitioning algorithms remains outside the scope of this paper. In the experimental section we generate the block structure with a supervariable agglomeration procedure available in MAGMA-sparse [12]. This generates a problem-optimal block diagonal structure where the distinct diagonal blocks may differ in size, and their dimension is bounded by some pre-defined maximum size. For convenience, we refer to this upper bound as the block size, but we recognize that some diagonal blocks may be of smaller dimension to better match the block structure of the target problem.

3.1 Batched Gauss-Jordan elimination

BGJE computes the inverses of a large set of small dense submatrices corresponding to the diagonal blocks D_i. The CUDA kernel for this purpose schedules one warp to handle the inversion of one block. This allows to leverage the large register count and the warp shuffle instructions supported by CUDA architectures of compute capability 3.0 and higher. Concretely, at the beginning of the kernel, each warp reads a block from main memory into registers, and handles the complete inversion process in registers. In addition, each

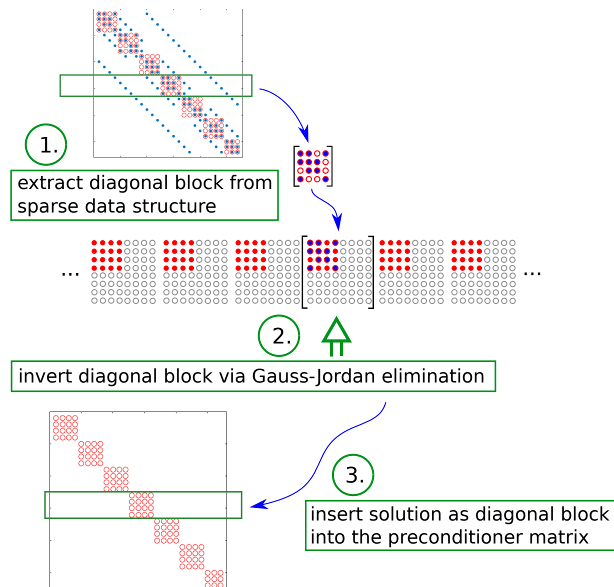


Figure 2. Generation of the block-Jacobi preconditioner via a set of batched routines: 1) data extraction; 2) BGJE; 3) data insertion. The block structure is indicated with red circles, the nonzero pattern of the system matrix with blue dots.

thread of the warp operates on a single row of the block, and the elements required by other threads are exchanged via warp shuffles.

Using implicit pivoting, as described in Section 2.3, we can avoid all row permutations during the inversion process. The row permutations, needed in the standard algorithm, are detrimental to performance as each pivot step leaves all threads idle except for those assigned to the two rows that are swapped. The implicit pivoting strategy accumulates all pivoting steps. This allows to realize them simultaneously at the end, when each thread writes its local row to the appropriate location of the inverse matrix.

As register arrays only support direct addressing, each index must be known at compile time to prevent the CUDA compiler from allocating the arrays in each thread's local memory, which shares the same physical space with global memory, and thus incurs the same access overhead. This is especially important to attain high performance on Maxwell and Pascal GPUs as, for these architectures, local memory requests are no longer cached in the on-chip L1 cache, but only in the off-chip L2, which is shared by all multiprocessors and much slower than the L1 [14].

As warp shuffles can only be used for communication within the same warp, the scope of the kernel is limited to (square) blocks with dimension $m \leq 32$. The idea of a block-Jacobi preconditioner is to map the size of the blocks D_i to the natural block structure of the system matrix, which origins, e.g., from a higher-order finite element discretization. As these blocks are usually of moderate size, and in a majority of cases contain less than 32 columns/rows, the algorithm covers the typical application area for block-Jacobi preconditioning. If the dimension of a block is less than 32, the remaining threads in the warp remain idle during the inversion step.

3.2 Batched data extraction and insertion

BGJE expects a collection of small dense blocks as input, while the block-Jacobi preconditioner needs to be generated for a sparse (coefficient) matrix stored in CSR format. Thus, a preprocessing step is needed to extract the diagonal blocks from the sparse data structure, and convert them into a set of dense matrices.

Use of caches. The extraction step can be implemented by instructing the threads of i-th warp to traverse the rows of the system matrix corresponding to D_i, keeping only the elements that belong to this diagonal block. This approach relies on the L1 and L2 caches to store the data needed for consecutive memory transactions in the extraction step. After completion of the GJE step, the inverse diagonal blocks are written into the preconditioner matrix stored in main memory. We refer to these strategies as *cached extraction* and *cached insertion*. Unfortunately, as the CSR format is based on row-major storage, this approach results in uncoalesced memory accesses; see Figure 3.

Use of shared memory. A coalescent alternative is possible by introducing an intermediate step that generates the dense matrices first in shared memory before writing them back to main memory. Here the threads of a warp traverse the elements in the corresponding rows of the sparse matrix structure, and store the elements part of the diagonal block as a dense matrix in column-major format in shared memory. The subsequent GJE step then benefits from the coalesced access to the dense system, and the access-friendly column-major order; see Figure 3. After completing the inversion, the reverse strategy inserts the inverse diagonal blocks into the CSR structure of the block-Jacobi preconditioner. We refer to these strategies as *shared extraction* and *shared insertion*.

3.3 Block-Jacobi generation

The generation of the block-Jacobi preconditioner comprises the three steps: data extraction from the sparse matrix A, inversion via BGJE, and the insertion of the inverse diagonal blocks into the sparse structure for the preconditioner $\hat{D}$. These steps can be realized as a sequence of three kernels, or merged into a single routine. Merging the distinct steps into one kernel makes the generation of the dense systems in main memory obsolete, and therewith significantly reduces the volume of accesses to main memory. However, the knowledge about the indexing that is necessary to avoid register spills during BGJE then transfers to the extraction step. This implies that all accesses to the dense matrix rows in the extraction and the insertion step must use indexing known at compile time.

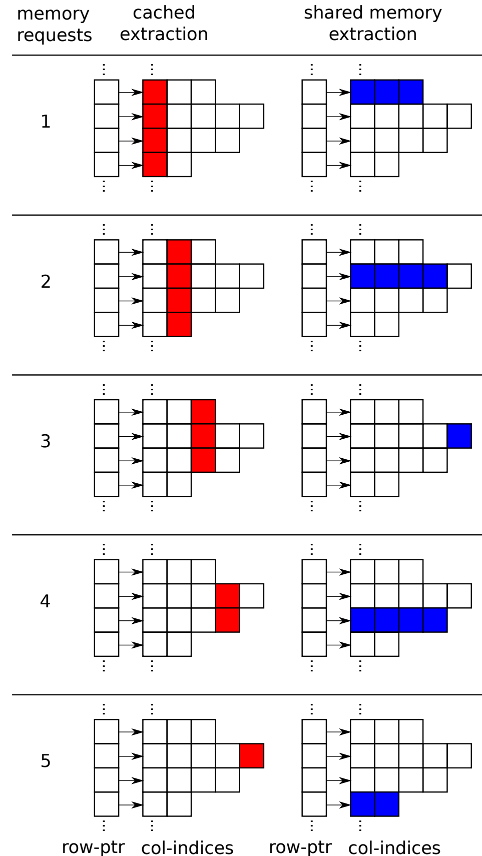


Figure 3. Illustration of the memory requests for the cached extraction and shared extraction (left and right, respectively). We assume warps of 4 threads, and visualize the data read by the distinct threads at each iteration with colored cells. We only show the accesses to the vector storing the col-indices of the CSR matrix structure; the access to the actual values induces far less overhead, as these memory locations are accessed only if a location belonging to a diagonal block is found. In that case, the access pattern is equivalent to the one used for col-indices.

4. Experiments

We next evaluate the performance of our block-Jacobi preconditioner generation kernel, described in Section 3, with a series of experiments designed to test different properties of the approach.

We begin by comparing the performance of the batched inversion step with the conventional (LU-based) batched inversion routine available in MAGMA 2.2.0 [12]. Next, we analyze the cost of the data extraction and insertion steps on a set of sparse matrices from the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection; see `http://www.cise.ufl.edu/research/sparse/matrices/` and the three leftmost columns in Table 2 for details.) Finally, we evaluate the benefits of the block-Jacobi preconditioner compared with the standard Jacobi in an iterative solver setting.

We conduct these experiments on a variety of different GPU architectures. This exposes the effect that various hardware features have on the behavior of different preconditioner generation strategies. The conclusions are therefore not tied to specific hardware, but cover a broad range of hardware designs.

4.1 Hardware and software framework

The GPUs used for the experiments belong to NVIDIA's Tesla series for high performance computing: K20, K40, K80, and P100. This covers the recent compute capabilities designed with full double-precision support: 3.5 (K20 and K40), 3.7 (K80) and 6.0 (P100). We exclude the outdated Fermi (compute capability 2.0) GPUs, as our routines rely on register shuffles and require large numbers of registers per thread, which are not available on this hardware. All computations use double precision arithmetic. Since the complete algorithm is executed on the GPU, the CPU in the host is not relevant for the following experimentation. We use NVIDIA's GPU compilers that ship with the CUDA toolkit 8.0.

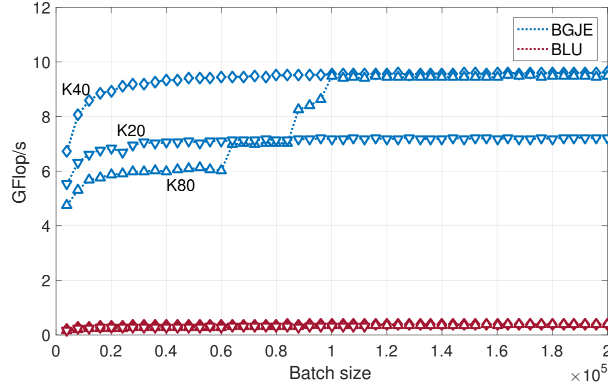
All kernels are implemented using the CUDA programming model and are designed to integrate into the MAGMA-sparse library [12]. MAGMA-sparse is also leveraged to provide a testing environment, the block-pattern generation, and the sparse solvers.

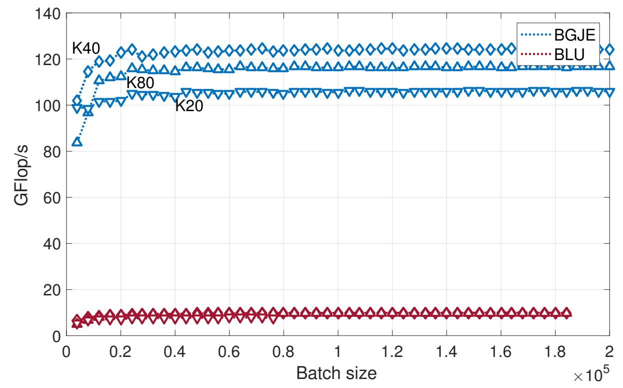
4.2 Performance of BGJE

Figure 4 compares the performances of our BGJE inversion routine with the LU-based alternative implemented in MAGMA. The two codes are not completely interchangeable, as the BGJE implementation works for matrices of row/column dimension m up to 32, while the LU-based inversion can handle also larger matrices, however all of them having the same size. For this reason, we limit this experiment to inputs that accommodate both constraints: all matrices in the batch are of the same size m and have at most 32 rows/columns. For brevity, we only show the results for $m = 8$

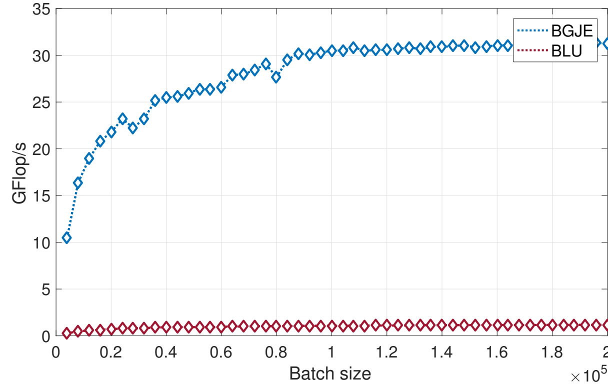
Block size 8	Block size 32

Kepler architecture





Pascal architecture



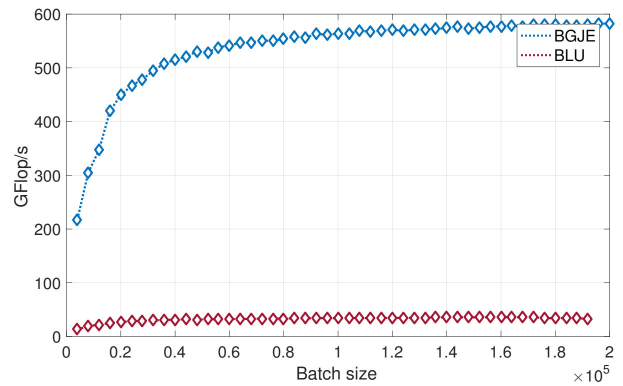


Figure 4. Performance of the batched matrix inversion routines: BGJE vs the batched LU (BLU) kernel in MAGMA. For the Kepler architecture, we distinguish the data for K20, K40, and K80 using triangles pointing down, diamonds, and triangles pointing up, respectively.

and 32. We remind that the inverse of a sparse matrix is, in general, dense. Thus, the cost of inverting a matrix via LU depends on the actual numerical values only to a minor extent (which is due to differences in the permutation sequence). Furthermore, due to the integration of implicit pivoting, the cost of our matrix inversion via GJE does not depend on the permutation sequence at all. Therefore, the actual data matrices that we used in the evaluation of this particular step is irrelevant.

The results in Figure 4 show the superiority of the GJE-based approach in terms of billions of floating-point operations (Flops) per second (GFlops/s). This can be attributed to better load balance of GJE in a parallel setting, use of registers for matrix storage, and decreased memory movement due to implicit pivoting. As a side note, a prototype implementation of the GJE-based approach which failed to have all register array indexing known at compile time achieved slightly better results than MAGMA's LU solution on Kepler GPUs, but offered worse performance on P100. As argued in Section 3, this is a side effect of architectural changes introduced on Maxwell GPUs, where local memory accesses are no longer cached in on-chip L1 cache. Additionally, even for the final kernel which is optimized to use registers instead of local memory, the compiler supplied with the older CUDA toolkit version 7.5 failed to produce a PTX code that uses registers. However, the new CUDA 8.0 compiler produced properly optimized PTX code.

4.3 Performance of data extraction+insertion

We perform the following experiments using four strategies to generate the block-Jacobi preconditioner:

1. *Block-Jacobi setup using cache* (UC).
2. *Merged block-Jacobi setup using cache* (MC).
3. *Block-Jacobi setup using shared memory* (US).
4. *Merged block-Jacobi setup using shared memory* (MS).

In particular, to reduce the number of possibilities, we exclude hybrid combinations that merge shared extraction with cached insertion or shared insertion with cached extraction.

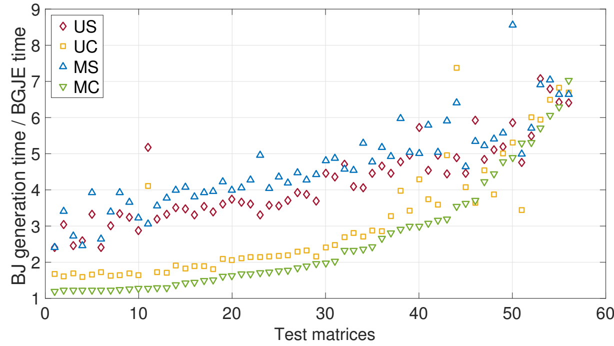
To compare the effect of different approaches, we measure the cost of data extraction+insertion as the percentage of runtime increase of the entire three-step block-Jacobi generation (extraction, BGJE, and insertion) compared to BGJE only. Since the amount of flops increases linearly with respect to the system matrix dimension, we can expect this overhead to be considerable as the performance is bounded by data movement.

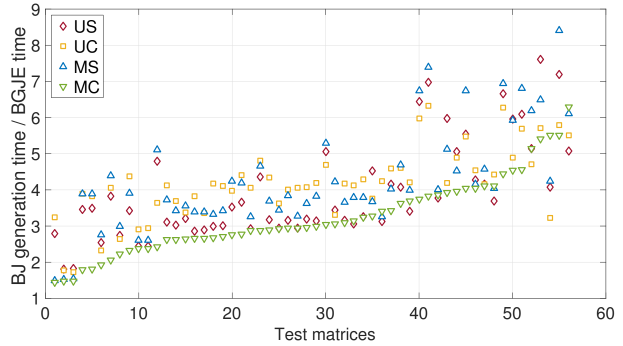
Figure 5 visualizes the costs of extraction+insertion for the different system matrices. The plots reveal several interesting insights: 1) No strategy is an overall winner. The cached versions usually work better for sparsity patterns with smaller block sizes, while the ones using shared memory are the preferred choice for larger block sizes. 2) The shared memory versions offer higher performance on P100 (and to some extent on K80) than on K40 and K20. This can be explained by inspecting the profiling results in Table 1. The shared memory versus register count ratios are higher in favor

Block size 8

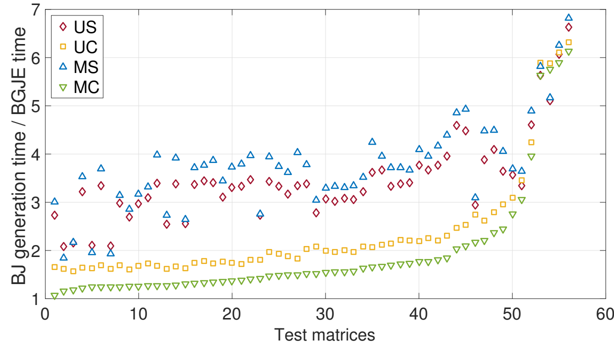
Block size 32

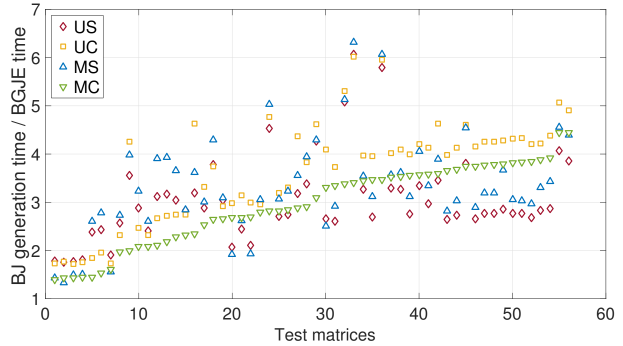
K20



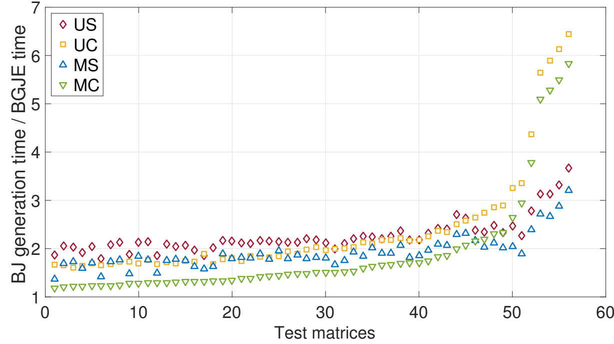


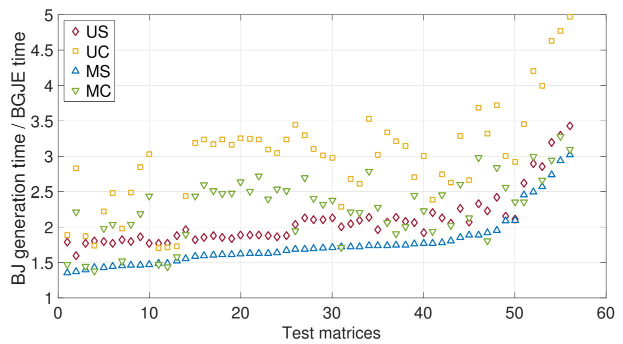
K40



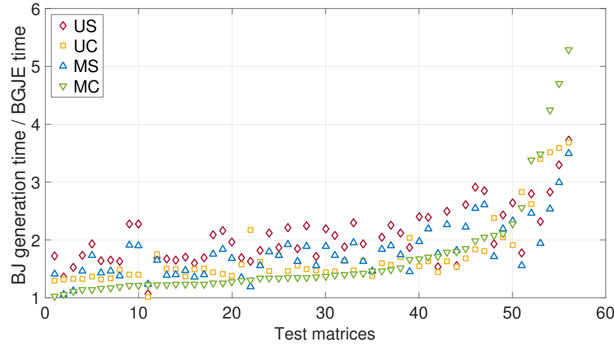


K80





P100



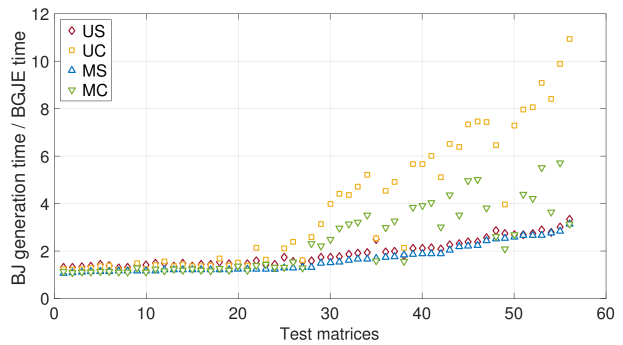


Figure 5. Time required to generate the block-Jacobi preconditioner (all three steps) relative to the execution time of the BGJE step. The matrices in each figure are sorted according to the fastest routine (on average) for that particular combination of block size and architecture.

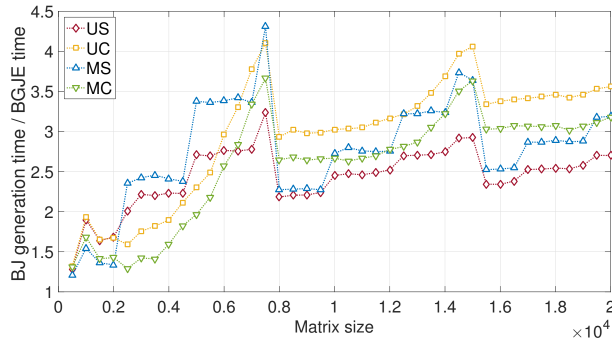
		K20/K40 (3.5)			K80 (3.7)			P100 (6.0)		
		smem	regs	blocks	smem	regs	blocks	smem	regs	blocks
	SM characteristics	48	65,536	16	112	131,072	16	64	65,536	32
Max blocks / SM	inversion	-	21	16	-	42	16	-	14	14
	extraction cache	-	146	16	-	292	16	-	146	32
	insertion cache	-	146	16	-	292	16	-	157	32
	merged cache	-	20	16	-	40	16	-	13	13
	extraction shared	6	113	6	14	227	14	8	136	8
	insertion shared	6	89	6	14	178	14	8	52	8
	merged shared	6	20	6	14	40	14	8	14	8

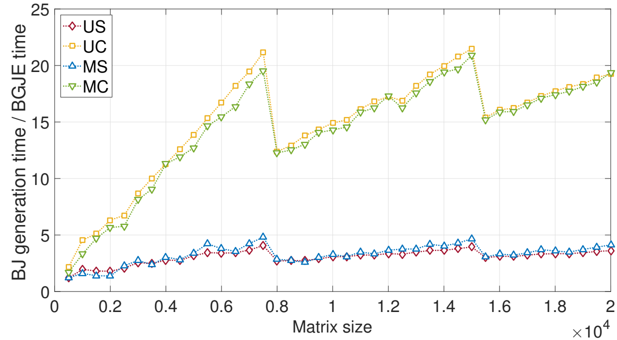
Table 1. Factors limiting multiprocessor occupancy for different kernels and architectures. The first row shows the shared memory (smem) in KB and the amount of registers (regs) available on each multiprocessor, as well as the maximal number of blocks that can be scheduled on a multiprocessor at the same time (blocks). The "Max blocks / SM" section shows how shared memory and register usage affects the maximum number of resident warps per multiprocessor. The blocks column combines these two limiting factors with the hardware limit on the number of blocks to obtain the actual limit on the number of blocks. We note that the K20 and K40 architectures contain 15 SMs, the K80 contains 2x15 SMs, and the P100 contains 60 SMs.

Tridiagonal structure

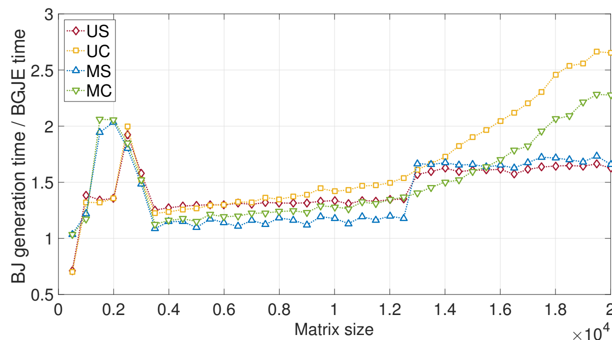
Arrow structure

K40





P100



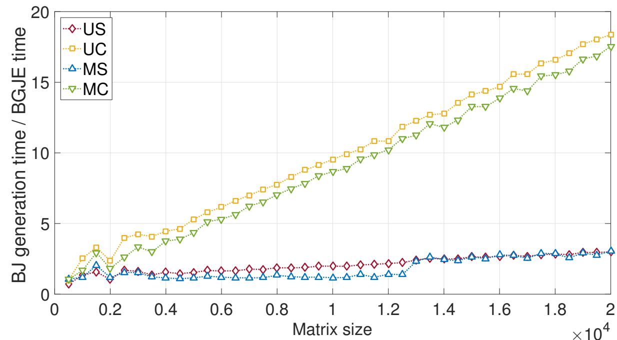


Figure 6. Runtime for generating the block-Jacobi preconditioner relative to the runtime of the BGJE inverting the diagonal blocks. The test matrices on the left have tridiagonal structure, the test matrices on the right side have arrow structure.

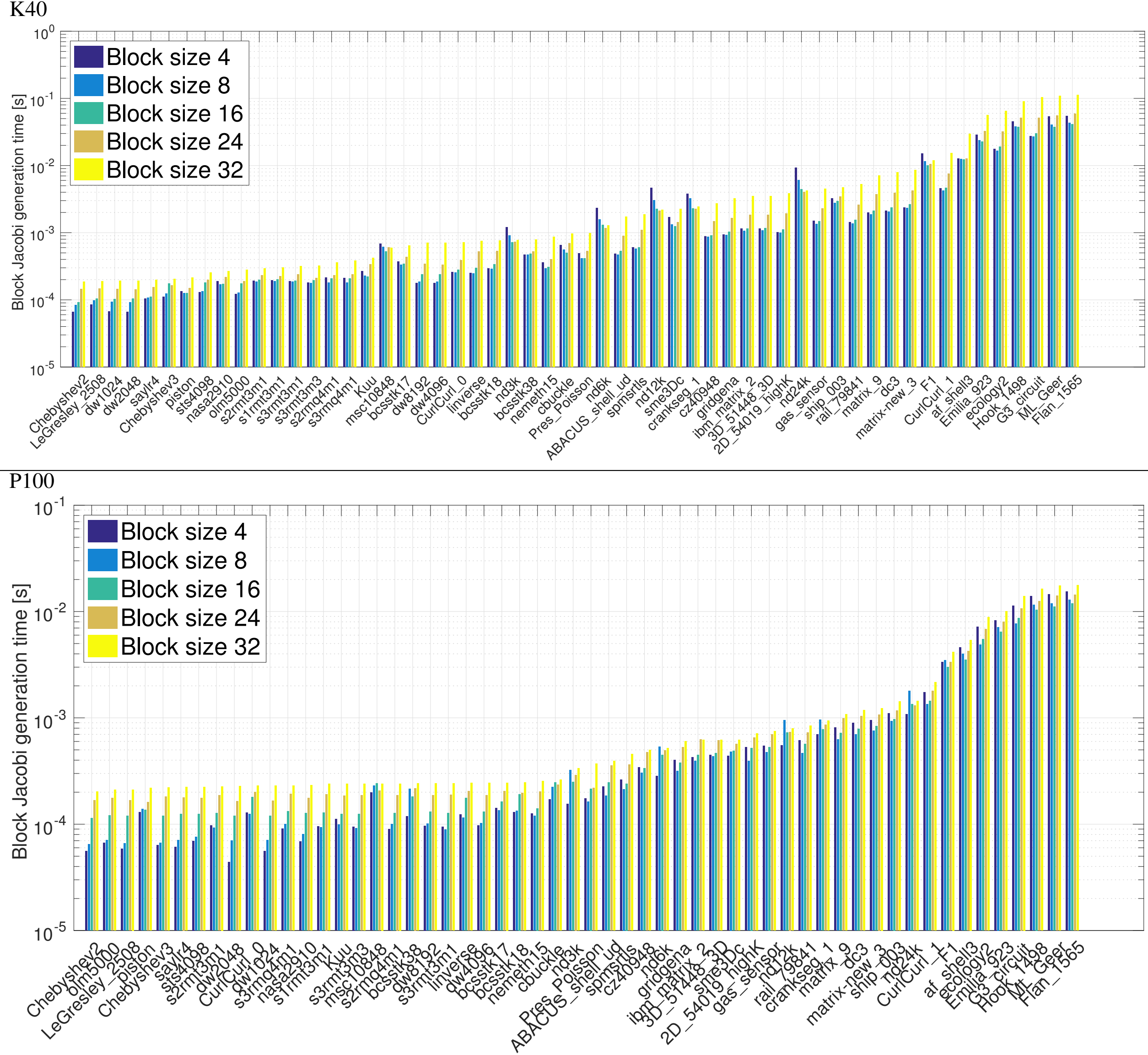


Figure 7. Runtime for generating the block-Jacobi preconditioner for different block sizes. For each block size we use the extraction strategy that gives the best average performance.

of shared memory for compute capabilities 3.7 and 6.0 than in 3.5. For the shared memory versions, the number of scheduled blocks per multiprocessor is limited by the shared memory consumption. Conversely, for the cache versions, the number of scheduled blocks per multiprocessor is limited by the register usage. As a result, on the older devices, the benefit of higher data locality cannot compensate for the lower occupancy. This also explains why US is faster than MS on older GPUs: When using the merged shared kernel, the shared memory requirement for extraction and insertion limits the number of blocks active on each SM. The unmerged version, however, separates the BGJE kernel from this restriction. While it uses the same number of blocks per SM for the extraction and insertion steps, it can schedule more blocks per SM for the inversion via BGJE, see Table 1.

Figure 6 demonstrates the effect which different sparsity patterns of the system matrix have on the performance of the extraction step. Concretely, we show the runtime of the block-Jacobi generation relative to only the inversion step for matrices with tridiagonal (left) and arrow (right) patterns. For the same size, the matrices have the same number of nonzeros, but differ in how the nonzeros locations are distributed. The different nonzero pattern have significant impact on the performance of the extraction strategies. For brevity, we only use a block size of 32 and only show the data for K40 and P100; the results for other block sizes and the K20 and K80 architectures are similar.

For the tridiagonal pattern, the extraction strategies using shared memory and the cache extraction+insertion achieve similar performance, as the memory access in the cache version of the extraction step is sufficiently coalescent for the matrices with a small number of nonzeros per row. Oppose to this, for the arrow sparsity pattern, the shared memory strategies are superior. For this pattern, the majority of nonzeros is located in the last row, and the extraction of

the last diagonal block becomes a bottleneck. UC and MC strategies allocate a single thread to extract data from this row, while US and MS distribute the computation required to extract the last block equally among all threads of the warp.

Therefore, we can expect both to provide comparable performance for balanced nonzero distributions, and the shared memory strategies to be superior for irregular patterns where few rows contain a large fraction of the nonzero elements.

4.4 Runtime of block-Jacobi generation

Figure 7 reports the total runtime of the block-Jacobi preconditioner generation for a few selected block sizes. As in Figure 6, we limit the data presented to the K40 and P100 architectures. We observe that the block-Jacobi preconditioner generation is in some cases faster for a larger block size than for a smaller block size. This comes from the fact that we always assign one warp to one diagonal block, and a smaller block size results in a higher block count.

4.5 Convergence benefits in the context of an iterative solver

Table 2 compares the convergence and execution time of an IDR(4) iterative solver [17] enhanced with either a (scalar) Jacobi preconditioner or a block-Jacobi preconditioner for the selected cases of the SuiteSparse collection. The execution time entails both the preconditioner generation and the iterative solver execution. All routines are taken from the MAGMA-sparse open source software package [12]. IDR is among the most robust Krylov solvers [4], and the IDR implementation available in MAGMA-sparse has proven to achieve performance close to the hardware-imposed bounds [5].

The results reveal that, for many of the test matrices, the scalar version of Jacobi fails to improve the convergence rate to meet the iteration limit of 100,000 iterations. For the test matrices where IDR(4) preconditioned with scalar Jacobi converges, the costlier block-Jacobi preconditioner generation is typically compensated by the more significant convergence improvement. This provides strong evidence that the overhead of the preconditioner generation remains small, and the block-Jacobi generation based on batched Gauss-Jordan elimination is an efficient tool in the context of iterative solvers on manycore architectures.

5. Summary and Future Work

We have proposed a batched method to assemble a block-Jacobi preconditioner on GPUs that exhibits a higher level of concurrency and can be expected to incur considerably less overhead than conventional ILU-type preconditioners. For the block-Jacobi preconditioner generation, we designed a batched Gauss-Jordan elimination kernel for the inversion of the diagonal blocks that strongly benefits from register use and an implicit pivoting strategy. We demonstrated that our batched Gauss-Jordan elimination outperforms the standard LU-based approach by more than an order of magnitude. Furthermore, we compared two efficient strategies for extracting the block diagonal of the sparse data structures storing the coefficient matrix, and concluded their performance being dependent on the GPU architecture. Finally, we demonstrated that the block-Jacobi preconditioned iterative solver is considerably more efficient than a solver enhanced with a simple scalar Jacobi preconditioner in terms of both, number of iterations for convergence and total runtime.

As part of future work, we will pursue additional performance improvements to the preconditioner application which leverage the block diagonal structure of the block-Jacobi matrix.

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 Number DE-SC-0010042. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO and FEDER.

References

[1] H. Anzt, E. Chow, and J. Dongarra. Iterative sparse triangular solves for preconditioning. In *Euro-Par 2015: Parallel Processing: 21st Int. Conf. on Parallel and Distributed Computing*, pages 650–661. Springer, 2015.

[2] H. Anzt, E. Chow, T. Huckle, and J. Dongarra. Batched generation of incomplete sparse approximate inverses on GPUs. In *Proc. 7th Workshop on Scalable Algorithms for Large-scale Systems*, ScalA'16, 2016.

[3] H. Anzt, E. Chow, D. Szyld, and J. Dongarra. Domain Overlap for Iterative Sparse Triangular Solves on GPUs. In H.-J. Bungartz, P. Neumann, and W. E. Nagel, editors, *Software for Exascale Computing - SPPEXA*, volume 113 of *Lecture Notes in Computer Science and Engineering*, pages 527–545. Springer International Publishing, 2016.

[4] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, and M. Koehler. Efficiency of General Krylov Methods on GPUs – An Experimental Study. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 683–691, 2016.

[5] H. Anzt, M. Kreutzer, E. Ponce, G. D. Peterson, G. Wellein, and J. Dongarra. Optimization and performance evaluation of the IDR iterative Krylov solver on GPUs. *Int. J. High Performance Computing & Applications*, 2016.

[6] P. Benner, P. Ezzatti, E. Quintana-Ortí, and A. Remón. Matrix inversion on CPU-GPU platforms with applications in control theory. *Concurrency and Computation: Practice and Experience*, 25(8):1170–1182, 2013. ISSN 1532-0634.

[7] E. Chow and A. Patel. Fine-grained parallel incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37(2):C169–C193, 2015.

[8] E. Chow and J. Scott. On the use of iterative methods and blocking for solving sparse triangular systems in incomplete factorization preconditioning. Technical Report Technical Report RAL-P-2016-006, Rutherford Appleton Laboratory, 2016.

[9] G. Golub and C. V. Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[10] A. Haidar, T. Dong, P. Luszczek, S. Tomov, and J. Dongarra. Batched matrix computations on hardware accelerators based on GPUs. *Int. J. High Performance Computing & Applications*, 29(2):193–208, 2015.

[11] A. S. Householder. *The Theory of Matrices in Numerical Analysis*. Dover, New York, 1964.

[12] Innovative Computing Lab. Software distribution of MAGMA version 2.0. `http://icl.cs.utk.edu/magma/`, 2016.

[13] J. Kurzak, H. Anzt, M. Gates, and J. Dongarra. Implementation and tuning of batched cholesky factorization and solve for NVIDIA GPUs. *IEEE Trans. on Parallel and Distributed Systems*, 27(7):2036–2048, July 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2015.2481890.

[14] NVIDIA Corporation. NVIDIA CUDA C programming guide. `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, September 2016. Version 8.0.

[15] E. S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R. van de Geijn. A note on parallel matrix inversion. *SIAM Journal on Scientific Computing*, 22(5):1762–1771, 2001.

[16] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

[17] P. Sonneveld and M. B. van Gijzen. IDR(s): A Family of Simple and Fast Algorithms for Solving Large Nonsymmetric Systems of Linear Equations. *SIAM Journal on Scientific Computing*, 31(2):1035–1062, 2009.

Matrix	size	#nnz/row	Jacobi		Block-Jacobi (4)		Block-Jacobi (8)		Block-Jacobi (16)		Block-Jacobi (32)	
			#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]
2D_54019_highK	–	–	7795	14.38	3083	5.85	1042	2.08	**301**	**0.63**	507	1.06
3D_51448_3D	–	–	–	–	–	–	7208	16.46	**214**	**0.52**	2505	5.86
ABACUS_shell_ud	2492	4.03	1764	2.96	1512	2.62	1919	3.42	1830	3.27	**1084**	**2.01**
af_shell3	1931	11.88	**1228**	**8.28**	1374	9.28	1066	8.32	1154	9.09	1108	9.46
bcsstk17	1271	2.17	1122	1.97	1141	2.01	609	1.11	**532**	**0.97**	574	1.06
bcsstk18	1038	1.48	590	0.99	504	0.90	360	0.66	**318**	**0.58**	311	0.60
bcsstk38	–	–	4772	9.17	3510	6.82	2009	3.97	**1402**	**2.78**	1820	3.55
cbuckle	207	0.41	103	0.23	73	0.15	63	0.15	**19**	**0.06**	49	0.11
Chebyshev2	–	–	–	–	138	0.41	47	0.13	34	0.10	**28**	**0.09**
Chebyshev3	–	–	–	–	–	–	148	0.58	80	0.32	**73**	**0.30**
crankseg_1	229	0.75	153	0.58	145	0.58	149	0.63	135	0.57	**103**	**0.47**
CurlCurl_0	94	0.14	72	0.14	64	0.13	52	0.12	50	0.11	**46**	**0.10**
CurlCurl_1	301	0.82	236	0.72	197	0.64	**130**	**0.51**	126	0.55	108	0.55
cz40948	–	–	63159	112.52	**33527**	**61.08**	–	–	39056	74.92	34785	66.82
dc3	237	24.51	143	14.77	**121**	**12.60**	124	12.87	138	14.43	189	19.69
dw1024	–	–	253	0.43	124	0.24	130	0.26	78	0.15	**52**	**0.10**
dw2048	–	–	253	0.45	124	0.25	130	0.26	78	0.15	**52**	**0.10**
dw4096	–	–	–	–	–	–	4713	7.30	1471	2.41	**876**	**1.45**
dw8192	–	–	–	–	–	–	4713	7.30	1471	2.44	**876**	**1.47**
ecology2	5985	37.78	4672	32.09	3231	23.90	2901	25.47	**2417**	**22.22**	2691	27.31
Emilia_923	–	–	50740	678.05	23572	329.08	51106	811.30	**17632**	**291.14**	42239	746.98
F1	2751	20.40	2255	15.82	2261	16.15	1928	15.17	1946	15.17	**1742**	**14.65**
Flan_1565	–	–	–	–	–	–	–	–	–	–	**155918**	**6130.28**
G3_circuit	2510	24.06	**974**	**10.00**	1091	13.90	1074	16.46	906	15.18	1028	18.97
gas_sensor	–	–	–	–	17310	37.17	11702	26.24	19057	43.12	**7724**	**17.65**
gridgena	1205	2.16	830	1.55	**662**	**1.26**	696	1.38	672	1.36	696	1.44
Hook_1498	–	–	29323	694.23	6394	143.86	6270	161.82	16930	671.64	**2684**	**71.45**
ibm_matrix_2	–	–	–	–	–	–	7007	16.02	**206**	**0.52**	2275	5.36
Kuu	98	0.16	74	0.13	68	0.14	69	0.13	59	0.13	**58**	**0.13**
LeGresley_2508	261	0.39	203	0.37	176	0.33	188	0.36	168	0.31	**153**	**0.28**
linverse	2761	4.17	–	–	–	–	6351	10.41	1986	3.37	**870**	**1.59**
matrix_9	1451	3.01	602	1.32	661	1.52	496	1.23	**89**	**0.29**	489	1.31
matrix-new_3	–	–	5270	11.43	5640	12.96	5277	13.05	**221**	**0.62**	2290	5.98
ML_Geer	–	–	1659	38.10	1345	37.28	2110	66.81	1518	50.09	**1169**	**36.68**
msc10848	–	–	–	–	–	–	–	–	11503	22.16	**9372**	**17.93**
nasa2910	587	0.85	562	0.94	495	0.83	385	0.68	435	0.74	**339**	**0.61**
nd12k	25417	85.50	10932	35.91	3487	11.95	2130	7.49	**1300**	**4.58**	1308	4.63
nd24k	38865	209.63	13382	73.13	4007	21.20	1886	10.31	1865	9.94	**1149**	**6.45**
nd3k	–	–	–	–	–	–	2928	5.97	**1961**	**4.04**	2494	5.08
nd6k	–	–	–	–	8259	20.19	4370	10.81	2906	7.27	**1847**	**4.64**
nemeth15	51	0.09	45	0.10	–	–	70	0.15	**27**	**0.05**	31	0.06
olm5000	–	–	344	0.58	179	0.35	**114**	**0.22**	139	0.25	165	0.31
piston	–	–	122	0.24	125	0.25	125	0.25	93	0.19	**93**	**0.18**
Pres_Poisson	188	0.33	149	0.31	114	0.24	91	0.18	**86**	**0.17**	74	0.17
rail_79841	987	1.84	952	1.86	817	1.66	679	1.46	691	1.53	**517**	**1.20**
s1rmt3m1	237	0.35	165	0.32	150	0.29	132	0.27	**129**	**0.27**	135	0.28
s2rmq4m1	613	0.92	307	0.56	229	0.42	218	0.42	**198**	**0.37**	199	0.39
s2rmt3m1	1124	1.73	372	0.65	267	0.49	211	0.39	**163**	**0.32**	193	0.37
s3rmq4m1	–	–	3977	6.31	1007	1.65	1277	2.12	**425**	**0.74**	718	1.19
s3rmt3m1	–	–	–	–	8565	13.56	1354	2.25	**410**	**0.70**	1117	1.88
s3rmt3m3	–	–	–	–	10497	16.51	1503	2.49	**651**	**1.12**	907	1.51
saylr4	1077	1.52	359	0.59	255	0.45	183	0.32	**120**	**0.24**	130	0.26
ship_003	8324	20.29	**2150**	**5.58**	2219	6.08	2501	7.43	2102	6.31	1799	5.65
sme3Dc	3455	7.91	3216	7.79	3449	8.53	3762	9.54	2680	6.92	**2405**	**6.22**
spmsrtls	–	–	–	–	–	–	–	–	39425	71.16	**7797**	**14.18**
sts4098	103	0.19	82	0.18	85	0.19	70	0.16	59	0.13	**54**	**0.12**

Table 2. Iterations and execution time of IDR(4) enhanced with scalar Jacobi preconditioning or block-Jacobi preconditioning. The runtime combines the preconditioner setup time and the iterative solver execution time.