Parallel Computing 000 (2018) 1–16

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Variable-size batched Gauss–Jordan elimination for block-Jacobi preconditioning on graphics processors

Hartwig Anzt[a,b], Jack Dongarra[b,c,d], Goran Flegar[e], Enrique S. Quintana-Ortí[e,*]

[a] Karlsruhe Institute of Technology, Germany
[b] Innovative Computing Lab (ICL), University of Tennessee, Knoxville, TN, USA
[c] Oak Ridge National Laboratory, USA
[d] School of Computer Science, University of Manchester, United Kingdom
[e] Dept. Ingeniería y Ciencia de Computadores, Universidad Jaume I, Castellón, Spain

ARTICLE INFO

Article history:
Received 6 April 2017
Revised 5 December 2017
Accepted 29 December 2017
Available online xxx

Keywords:
Batched algorithms
Matrix inversion
Gauss–Jordan elimination
Block-Jacobi
Sparse linear systems
Graphics processor

ABSTRACT

In this work, we address the efficient realization of block-Jacobi preconditioning on graphics processing units (GPUs). This task requires the solution of a collection of small and independent linear systems. To fully realize this implementation, we develop a variable-size batched matrix inversion kernel that uses Gauss-Jordan elimination (GJE) along with a variable-size batched matrix–vector multiplication kernel that transforms the linear systems' right-hand sides into the solution vectors. Our kernels make heavy use of the increased register count and the warp-local communication associated with newer GPU architectures. Moreover, in the matrix inversion, we employ an implicit pivoting strategy that migrates the workload (i.e., operations) to the place where the data resides instead of moving the data to the executing cores. We complement the matrix inversion with extraction and insertion strategies that allow the block-Jacobi preconditioner to be set up rapidly. The experiments on NVIDIA's K40 and P100 architectures reveal that our variable-size batched matrix inversion routine outperforms the CUDA basic linear algebra subroutine (cuBLAS) library functions that provide the same (or even less) functionality. We also show that the preconditioner setup and preconditioner application cost can be somewhat offset by the faster convergence of the iterative solver.

1. Introduction

Solving large, sparse-linear systems of equations is a prevailing problem in scientific and engineering applications that involve the discretization of partial differential equations (PDEs). A standard approach to tackle these problems combines a Krylov method with a preconditioner that accelerates the iterative solution process [1]. In the context of high-performance computing, the efficiency of the preconditioner depends on the parallel scalability of both the preconditioner generation (prior to the iterative solve) and the preconditioner application (at each step of the iterative solve).

Using preconditioners based on Jacobi (diagonal scaling) and block-Jacobi typically renders moderate improvements to the convergence of the iterative solver [1]. These acceleration techniques are nevertheless reasonably attractive as block-

[*] Corresponding author.
E-mail addresses: hanzt@icl.utk.edu (H. Anzt), dongarra@icl.utk.edu (J. Dongarra), flegar@uji.es (G. Flegar), quintana@uji.es, quintana@icc.uji.es (E.S. Quintana-Ortí).

https://doi.org/10.1016/j.parco.2017.12.006
0167-8191/© 2018 Elsevier B.V. All rights reserved.

diagonal scaling introduces very small computational overhead to the solver iteration. Furthermore, the application of a Jacobi-type preconditioner is inherently parallel and, therefore, highly appealing for massively parallel architectures.

In [2] we proposed a batched routine for the generation of a block-Jacobi preconditioner using the explicit inversion of the diagonal blocks. Precisely, we designed a variable-size batched routine for matrix inversion on graphics processing units (GPUs) based on Gauss–Jordan elimination (GJE) [3]. Furthermore, we introduced an implicit pivoting strategy in the GJE procedure that replaces row-swapping with a migration of the workload (i.e., operations) to the thread that owns the necessary data, which allowed us to realize the complete inversion process in thread-local registers. For the block-Jacobi preconditioner generation, the inversion process needs to be combined with routines that extract the diagonal blocks from the sparse data structure that stores the coefficient matrix. This extraction step can be costly, particularly for matrices with an unbalanced nonzero distribution. In response, we developed an extraction routine that balances coalescent data access, workload imbalance, and use of shared memory.

In this paper, we extend our previous work in [2] with new contributions, listed below.

- We propose a modified version of our variable-size batched GJE (BGJE) inversion routine for GPUs that can invert several blocks per warp. This avoids idle CUDA cores and operations on dummy values when processing a matrix batch where each matrix is less than or equal to 16×16.
- We introduce a new variant of the extraction procedure that requires a much smaller amount of shared memory. This strategy transposes the diagonal blocks at the time they are extracted from the sparse coefficient matrix, inverts the transposed diagonal block, and ultimately writes the inverse of the transpose in transposed mode. The result provides a functionality equivalent to the original block inversion but reduces the amount of shared memory in the block size, utilized during the inversion procedure, from quadratic to linear only.
- We replace the general sparse matrix–vector multiplication kernel in the preconditioner application with a specialized variant that exploits the block-diagonal structure of the preconditioner matrix. This accelerates the application of the block-Jacobi preconditioner in the iterative solution process.

Our results revealed that these modifications can render significant performance improvements, particularly when targeting batches consisting of small blocks like those appearing in block-Jacobi preconditioning for problems arising from finite element method (FEM) discretizations.

The rest of the paper is structured as follows. In Section 2 we offer a short review of Jacobi-type iterative solvers and batched routines for linear algebra. In Section 3 we further elaborate on the batched Gauss–Jordan elimination (BGJE) procedure presented in [2], and we describe the batched kernels and highlight the major improvements in the block-Jacobi generation step, extraction step, and preconditioner application. In Section 4 we report on our extensive evaluation of the new BGJE routine on NVIDIA's K40 and P100 GPUs. Particularly, we focus on the performance acceleration produced by the modifications of the original BGJE. Finally, in Section 5 we summarize our contributions and the insights gained from the experimental evaluation.

2. Background and related work

2.1. Block-Jacobi preconditioning

Consider the linear system $Ax = b$, with 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 block-Jacobi method partitions the entries of the coefficient matrix as $A = L + D + U$, where $D = (D_1, D_2, \ldots, D_N)$ contains a collection of blocks (of variable-size) located on the diagonal of A, while L and U comprise the entries of A below and above those in D, respectively. For a starting solution guess $x^{\{0\}}$, the iterative Jacobi solver can then be formulated as:

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

where the convergence is ensured if the spectral radius of the iteration matrix $M = I - D^{-1}A$ is smaller than one [1]. This occurs, for instance, in diagonally-dominant systems [1].

In case it is well-defined, the (block-)Jacobi matrix can be used as preconditioner, transforming the original system $Ax = b$ into either the left-preconditioned system

$$D^{-1}Ax = c \quad (= D^{-1}b), \tag{2}$$

or the right-preconditioned system

$$AD^{-1}y = b, \tag{3}$$

with $x = D^{-1}y$. Hereafter, we will consider the left-preconditioned case.

When integrated into a Krylov subspace-based solver, the application of a block-Jacobi preconditioner in (2) requires the solution of the block-diagonal linear system (i.e., a linear system for each block D_i). Alternatively, assuming the block-inverse matrix $\hat{D} = D^{-1}$ is available, the block-diagonal scaling in (2) can be realized in terms of a matrix-vector multiplication with

```
 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     % explicit 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;
```

Fig. 1. Simplified loop-body of the basic GJE implementation in Matlab notation using standard pivoting.

the inverse blocks $\hat{D}_i = D_i^{-1}, i = 1, 2, \ldots, N$. In general, pre-computing explicitly the block-inverse $\hat{D}$ during the preconditioner setup 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. On parallel architectures, it is possible to exploit the pairwise independence of the diagonal blocks in D by generating their individual inverses in parallel.

2.2. GJE for matrix inversion

GJE has been proposed in the last years as an efficient method for matrix inversion on clusters of multicore processors and many-core hardware accelerators [4,5]. In addition, in [2] we demonstrate the benefits of leveraging GJE for block-Jacobi preconditioning on GPUs. When combined with partial pivoting, GJE is as stable as matrix inversion via the LU factorization. At the same time, GJE avoids the workload imbalance that occurs in the LU-based approach due to computations with triangular factors.

The basic algorithm for matrix inversion via GJE consists of a loop that comprises a pair of vector scalings (SCAL) and a rank-1 update (GER); see Fig. 1. The unblocked version of the GJE algorithm in this figure, based on Level-2 BLAS operations, generally yields a poor exploitation of the memory hierarchy on current processors. However, this formulation can deliver good performance when computing the inverses of small matrices, like those that usually appear in the context of block-Jacobi preconditioning. Finally, the Level-2 BLAS version of GJE allows the integration of an implicit pivoting strategy, which dramatically reduces explicit data movements.

2.3. GJE with implicit pivoting

To ensure numerical stability, GJE needs to include a pivoting strategy. On parallel architectures, the row swaps required in the standard partial pivoting technique (Fig. 1, line 8) can be costly. This is particularly the case if the data is distributed row-wise among the processor cores. In this scenario, the two cores holding rows `k` and `ipiv` need to exchange their data, while the other cores remain idle. Although distributing the matrix column-wise resolves this problem, the load imbalance is then just shifted to the pivot selection (line 6). As a response, in [2] we presented an implicit pivoting procedure which avoids explicitly swapping data. Instead, it accumulates all swaps, and combines them when completing the GJE algorithm. The paradigm underlying implicit pivoting is to move the workload to the thread owning the data, instead of keeping the workload fixed to the thread index and reshuffling the data.

In standard GJE with explicit pivoting, the data required for operations performed on each row at iteration `k` (lines 12–15) is located only in that particular row and the current pivot row `ipiv` (which was swapped with row `k` at the beginning of the iteration). The operation applied on the distinct rows only depends on whether or not a certain row is the current pivot row. Concretely, if a row is the current pivot (i.e., it lies on position `k`) the operation involves diagonal scaling; otherwise, it requires the scaling of element `k` followed by an `AXPY` of the remaining elements. Hence, the actual order of the rows is not important during the application of the Gauss–Jordan transformations, and the swaps can be postponed until the algorithm is completed. This idea is illustrated in Fig. 2.

The selection of the pivot entry has to be modified when pivoting implicitly. In explicit pivoting, at iteration `k`, all previous pivots are located above the `k`th entry of the diagonal, and the potential pivot rows for the current iteration lie in positions `k:m`. When using implicit pivoting, none of the rows have been swapped, so we need to keep track of the previously chosen pivots. At step `k`, the next pivot is chosen among the set of rows that were not yet selected as pivots. In Fig. 2, the potential pivots are the entries in rows `i` with "`p(i)` =0" in lines 6–9.

```
% Input  : m x m nonsingular matrix block Di.
% Output : Matrix block Di overwritten by its inverse
p = zeros(1, m);
for k = 1 : m
    % implicit spivoting
    abs_elems        = abs(Di(:, k));
    abs_elems(p > 0) = -1; % exclude already pivoted rows
    [abs_ipiv, ipiv] = max(abs_elems);
    p(ipiv)          = k;

    % Jordan transformation
    d          =  Di(ipiv, k);
    Di(:,k)    =-[Di(1:ipiv-1,k); 0; Di(ipiv+1:m,k)] / d;% SCAL
    Di         =  Di + Di(:,k) * Di(ipiv,:);             % GER
    Di(ipiv,:) = [Di(ipiv,1:k-1), 1, Di(ipiv,k+1:m)] / d;% SCAL
end
% Undo permutations
Di(p,:) = Di(:,p);
```

Fig. 2. Simplified loop-body of the basic GJE implementation in Matlab notation using implicit pivoting.

Since implicit pivoting does not change the execution order of the operations applied or the numerical values, this variant of pivoting preserves the numerical properties of the algorithm.

2.4. Batched GPU routines

The qualifier “batched” identifies a procedure that applies the same operation to a large collection of data entities. In general, the subproblems (i.e., the data entities) are all small and independent, asking for a parallel formulation that simultaneously performs the operation on several/all subproblems in order to yield a more efficient exploitation of the computational resources. Batched routines are especially attractive in order to reduce the overall kernel launch overhead on GPUs, as they replace a sequence of kernel calls with a single kernel invocation. In addition, if the data for the subproblems is conveniently stored in the GPU memory, a batched routine can orchestrate a more efficient (coalesced) memory access.

In recent years, the development of batched routines for linear algebra operations has received considerable interest because of their application in machine learning, astrophysics, quantum chemistry, hydrodynamics, and hyperspectral image processing, among others. Examples of batched kernels for the dense BLAS appear in [6,7], and there exists a strong community effort on designing a interface standard for these routines [8]. Aside from block-Jacobi, the adoption of batched routines for efficient preconditioner generation has also been recently studied in the context of using approximate triangular solves for incomplete factorization preconditioning [2,9].

3. Design of CUDA kernels

In [2] we designed a set of routines for the generation and application of block-Jacobi preconditioners via variable-size BGJE. In this section we review the key concepts in [2], and introduce several improvements to further accelerate both the generation and application of the preconditioner.

The generation of an inversion-based block-Jacobi preconditioner can be decomposed into three distinct steps: (1) extraction of the diagonal blocks; (2) inversion of these blocks; and (3) insertion of the inverse blocks into the preconditioner matrix. We visualize these steps for non-uniform block sizes in Fig. 3. The three steps can be realized as three separate CUDA kernels, or in terms of a single kernel doing all steps in sequence. The experimental results in [2] suggest that, in general, merging all operations into a single kernel results in higher performance. A reason for this is the reduced memory transfer, as realizing the operations in a single kernel avoids the main memory accesses that are necessary to transfer data between separate kernels. In this paper we therefore focus on merged kernels for generating block-inverse matrices.

The question of how to identify a convenient block structure for a given coefficient matrix and an upper bound limiting the size of the diagonal blocks remains outside the focus of this paper. Here, for all experiments we use the supervariable blocking routine available in MAGMA-sparse [10].

3.1. Variable-size batched Gauss–Jordan elimination

The central operation in the generation of an inversion-based block-Jacobi preconditioner is the inversion of the diagonal blocks in D. These blocks are all square, of small dimension, and independent. In [2] we designed a variable-size BGJE routine that assigns one CUDA warp (a group of 32 threads) to invert each diagonal block. The kernel is launched on a grid with the number of warps covering the number of diagonal blocks. Within a warp, parallelism is realized by each thread handling one row of the diagonal block. This limits the scope of the kernel to matrix batches where no matrix is of dimension larger than 32. As blocks of larger dimension are rarely encountered in the context of block-Jacobi preconditioning, the variable-size batched GJE kernel perfectly fits this application scope [2].

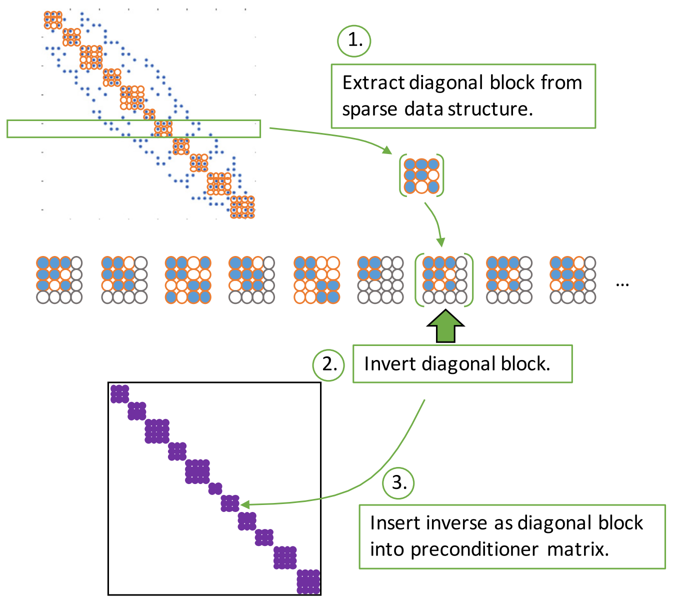
1.
Extract diagonal block from sparse data structure.
...
2.
Invert diagonal block.
3.
Insert inverse as diagonal block into preconditioner matrix.

Fig. 3. Generation of the block-Jacobi preconditioner: (1) data extraction; (2) variable-size BGJE; (3) data insertion. The block structure is indicated with orange circles, the original nonzero pattern with blue dots, and the block inverses with purple circles. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Handling the inversion with a single warp allows us to use two recent features of NVIDIA's GPU architectures: increased register count and warp shuffle instructions. In some detail, the data required by each thread (up to 32 data elements belonging to the matrix row) is first read into registers; the inversion is then computed using this data and communication occurs via the warp shuffle instruction (avoiding main memory access during the inversion process); and finally, the computed inverse is written back to main memory. In general, even though the diagonal blocks are sparse, their inverses are dense. We therefore handle and store the diagonal blocks in dense format during the complete inversion process.

The pivoting process ensuring numerical stability requires to identify the pivot element, see line 8 in Fig. 2. Since the matrix is distributed row-wise among the threads, this requires a parallel reduction. We realize this step via warp shuffles. The same type of shuffles is also used to distribute the contents of the current pivot row `Di(ipiv, k)` required for the operations in lines 12–15.

Multiple problems per warp. The variable-size BGJE presented in [2] assigns one warp to each diagonal block. In that work, diagonal blocks of size $k < 32$ were padded with dummy values to that dimension and the threads only execute the first k iterations of the GJE algorithm. Obviously, for small blocks, this wastes a significant part of the computational resources, as most of the threads then operate on dummy data. In this work, we improve the algorithm by allowing one warp to handle multiple small problems simultaneously. Concretely, let k_m denote the size of the largest block in the matrix batch and p_m stand for the smallest power of 2 such that $p_m \geq k_m$; then, in our new approach, each warp processes $32/p_m$ blocks. Proceeding in this manner, each group of p_m threads – we call it *sub-warp* – is assigned to one problem. The first k_m threads in the sub-warp compute the inverse of a block of size $k \leq k_m$ by padding it with dummy values to size k_m, and computing only the first k steps of the inversion procedure. The rest of the threads in the sub-warp remain idle. The reason for choosing p_m as the sub-warp size is that the CUDA ecosystem supports warp shuffles for these sizes. Using k_m instead of p_m would require additional operations to calculate the thread index. Finally, we do not consider "packing" blocks of different sizes into one warp (e.g., one warp could process blocks of sizes 15 and 17), as this would require a preprocessing step in order to determine which warp can process which set of blocks. Furthermore, it would also result in thread divergence between the two parts of the warp.

3.2. Data extraction from the sparse coefficient matrix

As BGJE expects a collection of small dense blocks as input, these blocks need to be extracted from the sparse coefficient matrix stored in CSR format [1]. We next review the two extraction strategies we implemented and compared in [2].

The first approach, named *cached extraction*, is a straight-forward method where each thread traverses a single matrix row, (specifically, the row whose values will be required by this thread during inversion process,) and extracts the elements that lie on the corresponding diagonal block. Since the CSR format is designed to favor accessing sparse matrix by rows, (i.e., it keeps the matrix entries in row-major order,) this will most likely result in non-coalescent memory access. Furthermore, an unbalanced nonzero distribution in the coefficient matrix inevitably incurs load imbalance, as threads operating on short

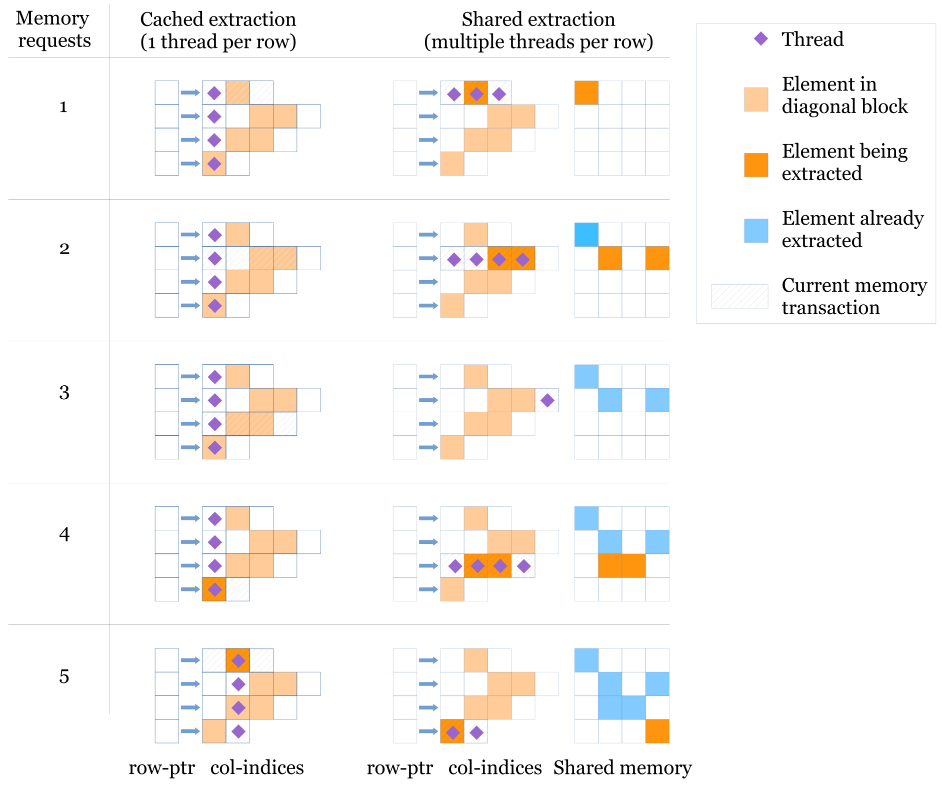
Memory requests
Cached extraction (1 thread per row)
Shared extraction (multiple threads per row)
1
2
3
4
5
row-ptr
col-indices
row-ptr
col-indices
Shared memory
Thread
Element in diagonal block
Element being extracted
Element already extracted
Current memory transaction

Fig. 4. Illustration of the memory requests for the cached extraction and shared extraction (left and right, respectively). We assume warps of 4 threads and memory transactions of 4 values. 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.

rows will remain idle while the remaining threads extract the data from their rows. Both effects impair the performance of the extraction step.

As a response to these issues, in [2] we proposed an alternative *shared extraction* method. The key idea is to eliminate non-coalescent memory access and (potential) load imbalance at the cost of using shared memory. Precisely, all threads of the warp collaborate on the extraction of the block by accessing each row containing part of the block in a coalesced mode (see Fig. 4). The diagonal block is then converted to dense format and stored into shared memory by writing the extracted values into the appropriate locations in shared memory, see right-hand side in Fig. 4. Once the extraction of a block is completed, each thread reads the values of the row assigned to it from shared memory. This strategy makes all memory accesses coalescent and alleviates load imbalance. The shared memory usage, however, can constrain the number of warps active per multiprocessor. On “older” GPU architectures we observed that the shared extraction strategy can result in lower performance due to this issue.

Reduced usage of shared memory. We improve the situation by radically reducing the amount of shared memory employed in the shared extraction step. This is possible because the inverse of a matrix A can be computed by first obtaining the inverse of A^T and then transposing the result (i.e., $((A^T)^{-1})^T = A^{-1}$). Extracting the transpose of the diagonal block is much easier as the ith elements of all columns are available as soon as the ith row is extracted to shared memory. This means that all threads can already read the ith row-value of the transposed block into registers before proceeding with the extraction of the next row. Thus, the extraction of the transpose block from the sparse matrix structure can be “interleaved” with the retrieval of the values from shared memory into the registers. As a result, the same shared memory locations used to store row k of the diagonal block can be re-used in the following step of the extraction. This reduces the total amount of shared memory required to that necessary to keep a single row of the diagonal block.

In case multiple blocks are assigned to each warp, a straight-forward extension of the strategy is to let each sub-warp extract the block assigned to it. This would, however, result in non-coalesced memory access. Coalesced memory access can be preserved by extracting all blocks handled by the warp in sequence, using all threads of the warp and enough shared memory to store one row of the largest matrix in the batch.

After the inversion of a diagonal block is completed, the result is written back to main memory. Realizing the afore-described extraction step in reverse order, we store the inverse of the transposed block in transposed mode – which is the inverse of the original block.

3.3. Preconditioner application

Once the block-inverse is generated, the block-Jacobi preconditioner can be applied in terms of a sparse matrix-vector product (SPMV).

Structure-aware SPMV. We improve the performance of the block-Jacobi preconditioner by replacing the generic sparse matrix-vector product with a specialized kernel that exploits the block structure of the preconditioner matrix. In detail, we use a variable-size batched dense matrix-vector multiplication (GEMV) to multiply the distinct block inverses D_i^{-1} with the appropriate vector segments. As in the preconditioner generation, the blocks are distributed among the (sub-)warps, with each (sub-)warp handling the multiplication for one vector segment. In contrast to the elements of the matrix, which are all used for only a single multiplication, the elements in the vector segment are reused in the multiplication with the distinct rows of the block. Hence, it is beneficial to read the elements of the vector segment into the registers of the distinct threads of the (sub-)warp (one element per thread) at the beginning of the routine. The performance of GEMV is constrained by the memory bandwidth. It is therefore essential to ensure coalesced memory accesses by forcing each (sub-)warp to read the diagonal blocks row-wise (the block-diagonal matrix is stored in the row-major based CSR matrix format). For each row of the block, the threads of the (sub-)warp then use warp shuffles to compute the dot product of the matrix row and the vector entries they keep in registers. Finally, the result is written to the appropriate position in the output vector.

4. Experimental evaluation

In this section, we (1) benchmark gigaFLOPS enhanced version of the variable-size batched matrix inversion routine based on GJE; (2) analyze the performance of the complete block-Jacobi preconditioner (generation and application); and (3) assess the efficiency of block-Jacobi preconditioning in an iterative solver setting.

We begin by comparing the new BGJE kernel against the version presented in [2] and against two batched inversion routines available in NVIDIA's cuBLAS library: `getriBatched` and `matinvBatched`. We note that both cuBLAS routines can only operate on batches of problems where all matrices are of the same size.

Next, to evaluate the performance benefits for the block-Jacobi preconditioner generation stage, in isolation, we combine our variable-size BGJE routines with the improved extraction and insertion procedures, and we test the block-inverse generation for different sparsity structures and block sizes. For this purpose, we consider a set of test matrices from the SuiteSparse Matrix Collection[1] (formerly known as the University of Florida Sparse Matrix Collection). In addition to the preconditioner generation, we also compare the specialized block-Jacobi application kernel based on variable-size batched GEMV with the generic SPMV routine from MAGMA-sparse [10].

Finally, to analyze the practical effects of the block-Jacobi preconditioning on an iterative solver, we integrate the block-Jacobi preconditioner into an Induced Dimension Reduction Krylov solver with shadow space dimension 4 (IDR(4)) and demonstrate the time-to-solution improvements obtained by replacing a scalar-Jacobi preconditioner with a block-Jacobi variant.

4.1. Hardware and software framework

For the experiments, we use the two most recent NVIDIA GPU architectures, which have full support for double-precision computations: the Kepler K40 (Compute Capability 3.5) and the Pascal P100 (Compute Capability 6.0). We do not consider the older Fermi and Maxwell architectures, as the former lacks support for warp shuffle instructions, and the latter does not implement full double-precision support. Because the batched matrix inversion routines, the block-Jacobi generation kernel, and the iterative solvers proceed exclusively on the GPU, details about the node's broader hardware specifications are irrelevant in the following experiments.

Our kernels are implemented using CUDA 8.0 and are designed to be integrated into the MAGMA-sparse library [10]. MAGMA-sparse also provides the testing environment, the block-pattern generation, and the sparse solvers used in our experiments. All computations use double-precision arithmetic—the standard in linear algebra.

4.2. Batched matrix inversion

This section analyzes the performance of four batched routines for matrix inversion on GPUs; `BGJE`, `BGJE-MPW`, `getriBatched`, and `matinvBatched`.

1. `BGJE` is the variable-size BGJE inversion kernel from [2].
2. `BGJE-MPW` is the enhanced kernel that incorporates the BGJE improvements described in this paper.
3. `getriBatched` renders the batched matrix inversion using two functions from NVIDIA's cuBLAS library: (1) `getrfBatched` computes the LU factorization of the matrix batch, then (2) `getriBatched` obtains the inverses using the results of the previous routine. All matrices in the batch are required to be of the same size.

[1] Visit http://www.cise.ufl.edu/research/sparse/matrices/.

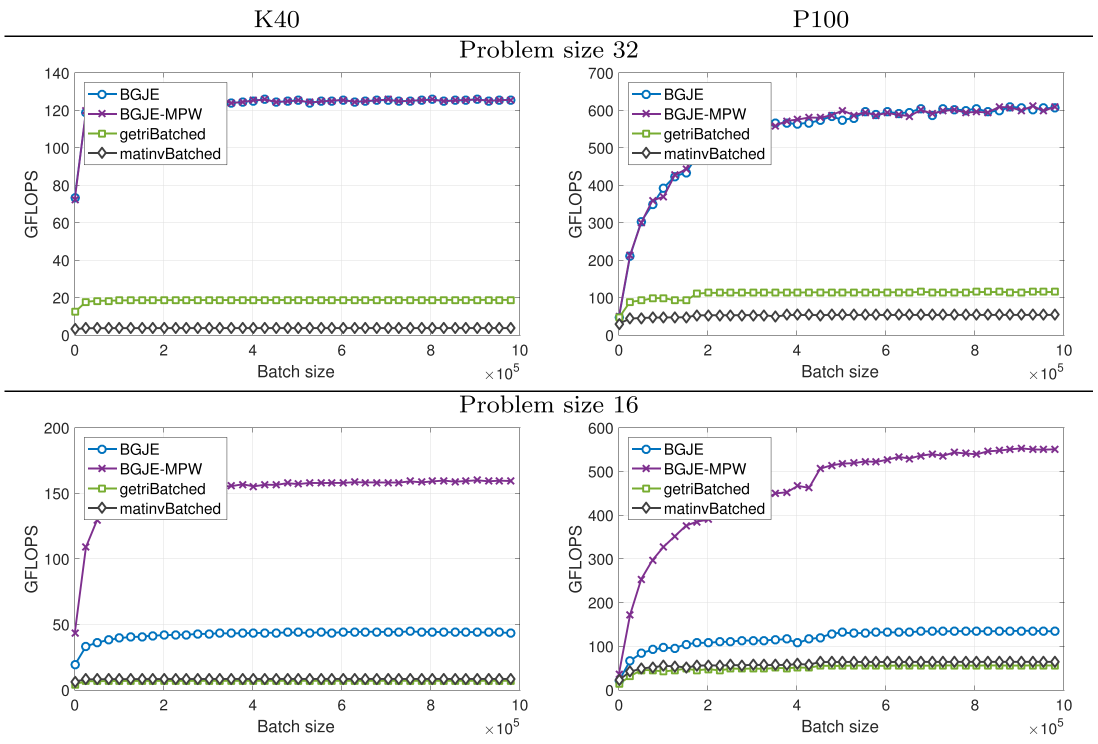

Fig. 5. Performance comparison of batched matrix inversion routines for various batch sizes. Top row shows matrices of size 32 × 32 and the bottom row shows matrices of size 16 × 16.

4. `matinvBatched` is NVIDIA's routine that merges the two calls of the `getriBatched` routine into a single kernel. Its functionality is limited to operating on batches of equal-size matrices with an upper bound of 32 × 32.

We note that the scope of the distinct batched inversion routines is slightly different: `BGJE`, `BGJE-MPW`, and `matinvBatched` only support matrices of size up to 32 × 32; and neither `matinvBatched` nor `getriBatched` support batches containing matrices of different sizes. Therefore, we limit the performance comparison to batches composed of equal-size matrices of up to 32 × 32. While this upper bound is usually not a problem in the context of block-Jacobi preconditioning, handling batches that contain variable-size matrices is essential to accommodating the inherent block structure of FEM discretizations. Consequently, the cuBLAS routines will not be considered in the complete preconditioner generation and application experiments.

Fig. 5 compares the performance, in terms of gigaFLOPS (billions of arithmetic floating-point operations per second), for two fixed matrix sizes (32 × 32 and 16 × 16) while increasing the matrix count (batch size). In a case where the matrix order is 32, both `BGJE` and `BGJE-MPW` deliver the same performance because, in this scenario, `BGJE-MPW` also schedules a single problem per warp. For this matrix size, the performance of both variable-size BGJE routines exceeds 600 gigaFLOPS (13% of the theoretical peak) on P100 and around 125 gigaFLOPS (9% of peak) on K40. These rates correspond to a 6× speedup over the batched inversion using `getriBatched`, and at least a 12× speedup over `matinvBatched`.

The older K40 architecture has a significantly lower register-per-core ratio compared to the P100. Because our `BGJE` and `BGJE-MPW` routines make heavy use of registers, a reduced register count limits the number of threads/warps that can be active on a multiprocessor, which explains the large performance gap between the K40 and P100 GPUs.

The two graphs on the left side of Fig. 5 clearly show that the registers are indeed a performance bottleneck on the K40. For batched problems consisting of 16 × 16 matrices, each thread only utilizes 16 registers (instead of 32 registers for 32 × 32 matrices), allowing more active threads—and therefore more active warps—per multiprocessor. As a result, the `BGJE-MPW` kernel delivers about 160 gigaFLOPS for the smaller matrix sizes but only around 125 gigaFLOPS for the larger matrices. In comparison, the `BGJE` kernel, which can only handle a single problem per warp, achieves a scant 40 gigaFLOPS for the small case. Moreover, both cuBLAS batched inversion routines, `getriBatched` and `matinvBatched`, deliver a meager 8 gigaFLOPS for this problem.

Again, note that the `BGJE` kernel pads the matrices with dummy elements to size 32 × 32 and inverts one system per warp. On the P100, this delivers less than 150 gigaFLOPS for a batch composed of matrices of size 16 × 16. In contrast, the performance of the `BGJE-MPW` routine exceeds 550 gigaFLOPS in similar conditions. Thus, although the performance of `BGJE-MPW` is lower for a 16 × 16 matrix than for a 32 × 32 matrix (which was expected because the data-movement-

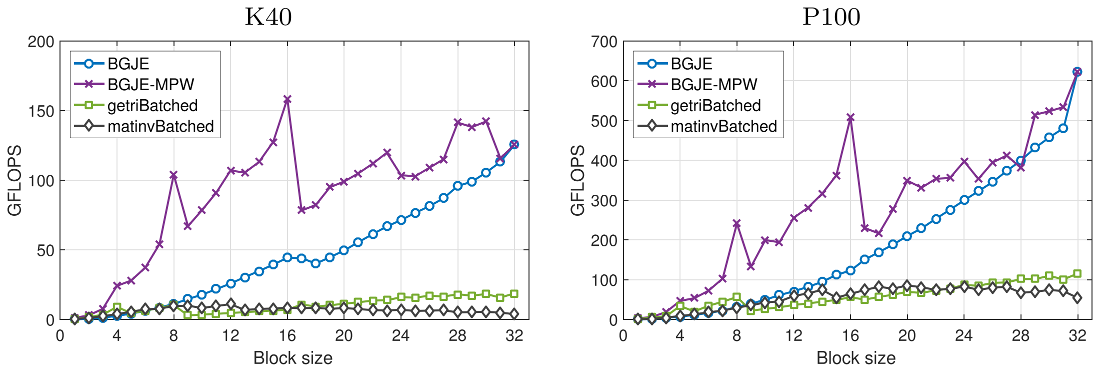

Fig. 6. Performance comparison of batched matrix inversion routines for various matrix sizes.

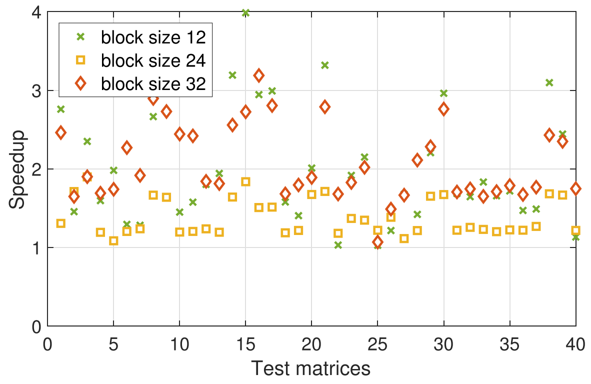

Fig. 7. Performance improvement from reducing the shared memory size in the block-Jacobi generation using shared extraction/insertion.

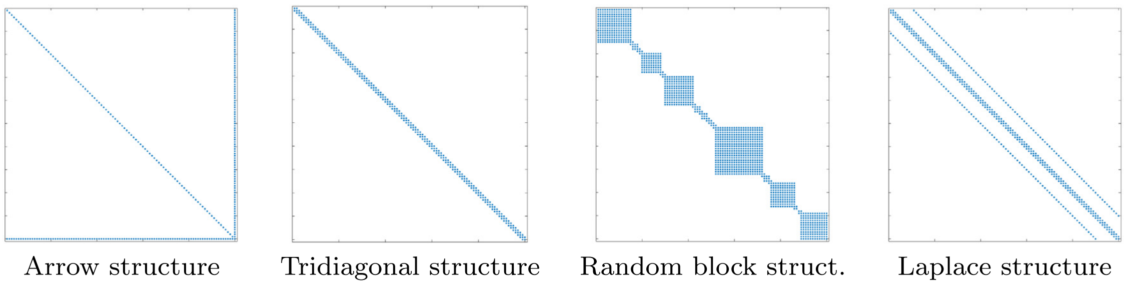

Fig. 8. Sparsity plots of test matrices used to evaluate the diagonal block extraction.

to-floating-point-operation ratio grows with the matrix size), BGJE-MPW is about one order of magnitude faster than the matrix inversion functions provided in NVIDIA's cuBLAS.

A detailed analysis for different matrix sizes is given in Fig. 6. In this experiment we fixed the batch size to 500,000 matrix problems and varied the dimension of the matrices in the batch from 1×1 to 32×32. For both architectures, BGJE exhibits a superlinear performance drop as the matrix size is reduced. This is because, for a batch with matrices of size k, each warp performs $2k^3$ useful operations, while the total volume of operations (including those on dummy data used for padding) is $2k \times 32^2$. In contrast, BGJE-MPW avoids most dummy operations and experiences only a linear performance loss—owing to inactive threads—between consecutive powers of two. Peaks for 16×16 matrices and 8×8 matrices clearly mark the thresholds where multiple small problems can be handled by a single warp without introducing any computational overhead. The performance lines for BGJE-MPW are more erratic than those observed for the other routines. The reason is that BGJE-MPW is implemented using C++ templates to generate a specialized version of the kernel for each matrix size. While this approach succeeds in minimizing the register count and the number of operations performed by the size-specific kernels, the kernel-specific resource requirements impact the number of warps that are active per multiprocessor and, ultimately, the kernel-specific performance.

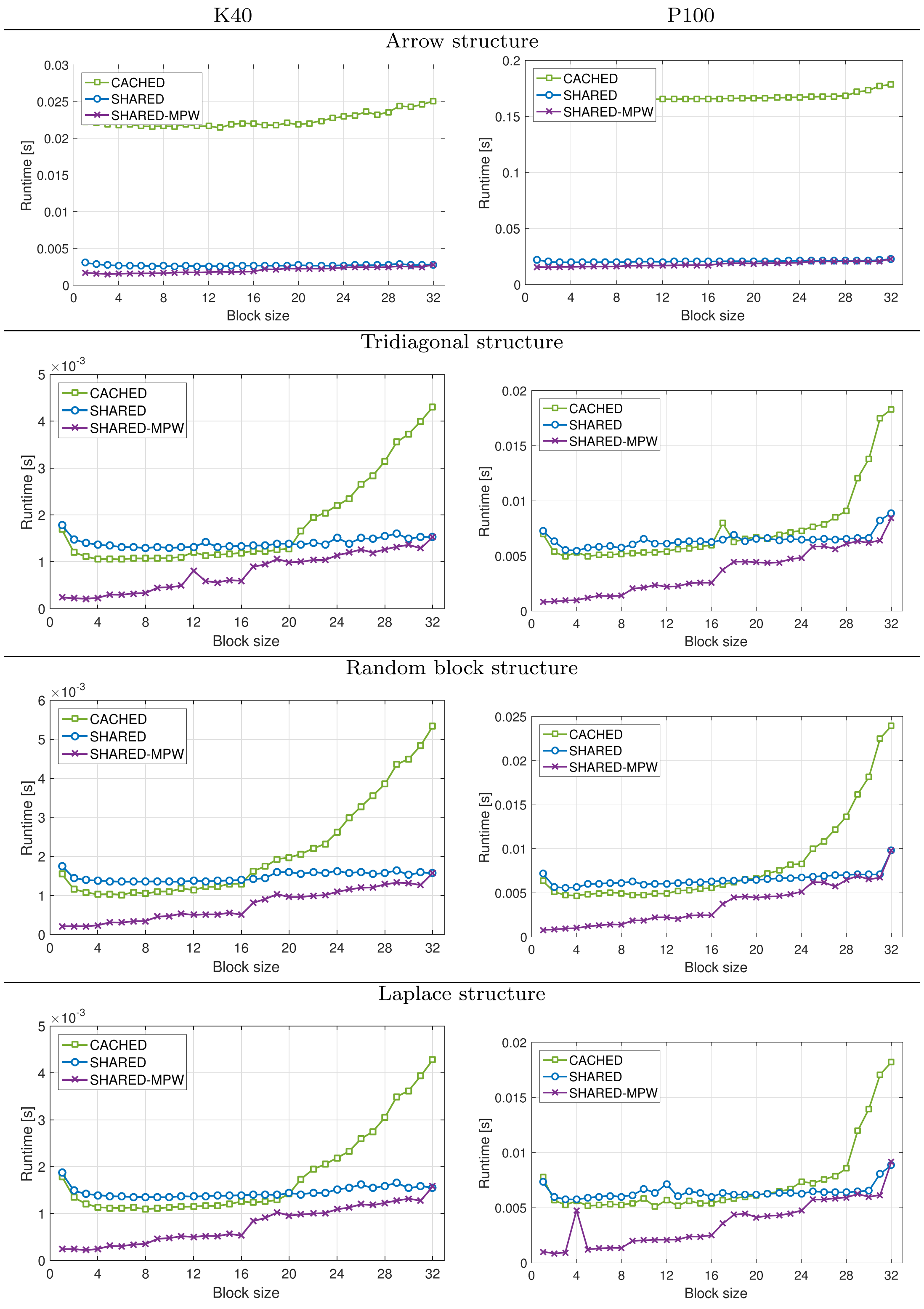

Fig. 9. Block-Jacobi generation time for increasing block sizes and nonzero distributions from top to bottom: arrow, tridiagonal, random block and Laplace.

Table 1
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.

Matrix	size	#nnz	ID	Jacobi		Block-Jacobi (8)		Block-Jacobi (12)		Block-Jacobi (16)		Block-Jacobi (24)		Block-Jacobi (32)	
				#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]	#iters	time [s]
ABACUS_shell_ud	23,412	218,484	26	3703	5.09	**2305**	**3.32**	2829	4.09	3028	4.43	2858	4.27	2418	3.70
bcsstk17	10,974	428,650	6	1967	2.87	1174	1.78	901	1.38	792	1.23	**735**	**1.20**	879	1.40
bcsstk18	11,948	149,090	39	1491	1.90	933	1.37	653	0.98	591	0.92	**440**	**0.65**	532	0.85
bcsstk38	8032	355,460	10	–	–	–	–	2459	4.21	4290	7.33	**1878**	**3.26**	2050	3.62
cbuckle	13,681	676,515	9	434	0.81	118	0.22	**48**	**0.11**	102	0.23	49	0.12	75	0.15
Chebyshev2	2053	18,447	5	–	–	197	0.53	62	0.18	53	0.15	38	0.11	**35**	**0.10**
Chebyshev3	4101	36,879	17	–	–	–	–	194	0.67	177	0.66	113	0.40	**77**	**0.31**
dc3	116,835	766,396	28	**168**	**17.29**	169	17.36	203	20.93	203	20.92	320	33.00	183	18.87
dw1024	2,048	10,114	30	–	–	169	0.23	148	0.27	163	0.30	92	0.18	**65**	**0.13**
dw2048	2,048	10,114	7	–	–	169	0.23	148	0.27	163	0.30	92	0.17	**65**	**0.12**
dw4096	8,192	41,746	18	–	–	–	–	–	–	6949	8.96	1732	2.33	**1012**	**1.46**
dw8192	8,192	41,746	36	–	–	–	–	–	–	6949	8.94	1732	2.35	**1012**	**1.41**
F1	343,791	26,837,113	27	3832	23.93	–	–	2460	16.34	2511	17.00	**1895**	**13.23**	2062	14.78
G3_circuit	1,585,478	7,660,826	31	2346	19.94	**2069**	**19.52**	2220	22.14	1935	20.09	2085	24.10	2198	26.83
gridgena	48,962	512,084	32	2265	3.51	1306	2.08	1766	2.84	1431	2.33	**1028**	**1.70**	1311	2.26
ibm_matrix_2	51,448	537,038	21	–	–	–	–	**227**	**0.46**	8992	17.45	254	0.55	2965	6.04
Kuu	7,102	340,200	3	162	0.29	103	0.19	95	0.18	**84**	**0.16**	94	0.18	84	0.17
LeGresley_2508	2,508	16,727	20	237	0.40	247	0.36	203	0.35	–	–	185	0.33	**166**	**0.31**
linverse	11,999	95,977	35	6185	7.80	–	–	–	–	6685	9.51	2175	3.07	**933**	**1.44**
matrix_9	103,430	1,205,518	2	1512	2.72	727	1.37	95	0.21	598	1.18	**87**	**0.20**	558	1.16
nasa2910	2,910	174,296	12	738	1.03	529	0.73	509	0.74	648	0.94	435	0.67	**392**	**0.64**
nd12k	36,000	14,220,946	19	–	–	6869	20.67	3183	9.67	2764	8.45	**1543**	**4.81**	1693	5.35
nd24k	72,000	28,715,634	11	–	–	4918	23.29	2906	13.85	2858	13.69	1916	9.35	**1457**	**7.16**
nd3k	9,000	3,279,690	33	–	–	–	–	4551	8.05	8270	14.55	**2640**	**4.78**	3196	5.87
nd6k	18,000	6,897,316	29	–	–	–	–	5142	11.14	9178	19.94	**2589**	**5.76**	2573	5.81
nemeth15	9,506	539,802	34	94	0.17	–	–	–	–	144	0.27	50	0.10	**49**	**0.10**
olm5000	5,000	19,996	15	–	–	1164	1.58	1049	1.44	256	0.41	545	0.80	**178**	**0.33**
Pres_Poisson	14,822	715,804	4	199	0.38	130	0.24	129	0.26	113	0.23	93	0.19	**82**	**0.17**
rail_79841	79,841	553,921	1	995	1.62	909	1.56	1013	1.78	**880**	**1.52**	862	1.58	810	1.52
s1rmt3m1	5,489	217,651	22	296	0.43	191	0.32	**171**	**0.24**	159	0.28	148	0.28	148	0.27
s2rmq4m1	5,489	263,351	37	708	0.97	231	0.39	**223**	**0.36**	262	0.43	214	0.36	198	0.36
s2rmt3m1	5,489	217,681	25	1016	1.35	327	0.50	209	0.35	218	0.36	**178**	**0.32**	220	0.40
s3rmq4m1	5,489	262,943	24	–	–	1387	1.95	599	0.89	1969	2.83	**515**	**0.80**	972	1.47
s3rmt3m1	5,489	217,669	16	–	–	–	–	693	0.99	2637	3.62	**529**	**0.75**	1142	1.71
s3rmt3m3	5,357	207,123	23	–	–	–	–	1995	2.74	2087	2.86	2229	3.14	**784**	**1.18**
saylr4	3,564	22,316	38	1907	2.46	387	0.59	246	0.40	281	0.38	**163**	**0.30**	170	0.32
ship_003	121,728	3,777,036	13	–	–	2058	4.89	1927	4.62	2849	6.97	**1683**	**4.26**	2160	5.75
sme3Dc	42,930	3,148,656	8	**2680**	**5.66**	4953	10.59	3101	6.74	3014	6.53	3566	7.92	4990	11.19
sts4098	4,098	72,356	14	135	0.26	113	0.23	78	0.12	94	0.19	76	0.16	**64**	**0.11**

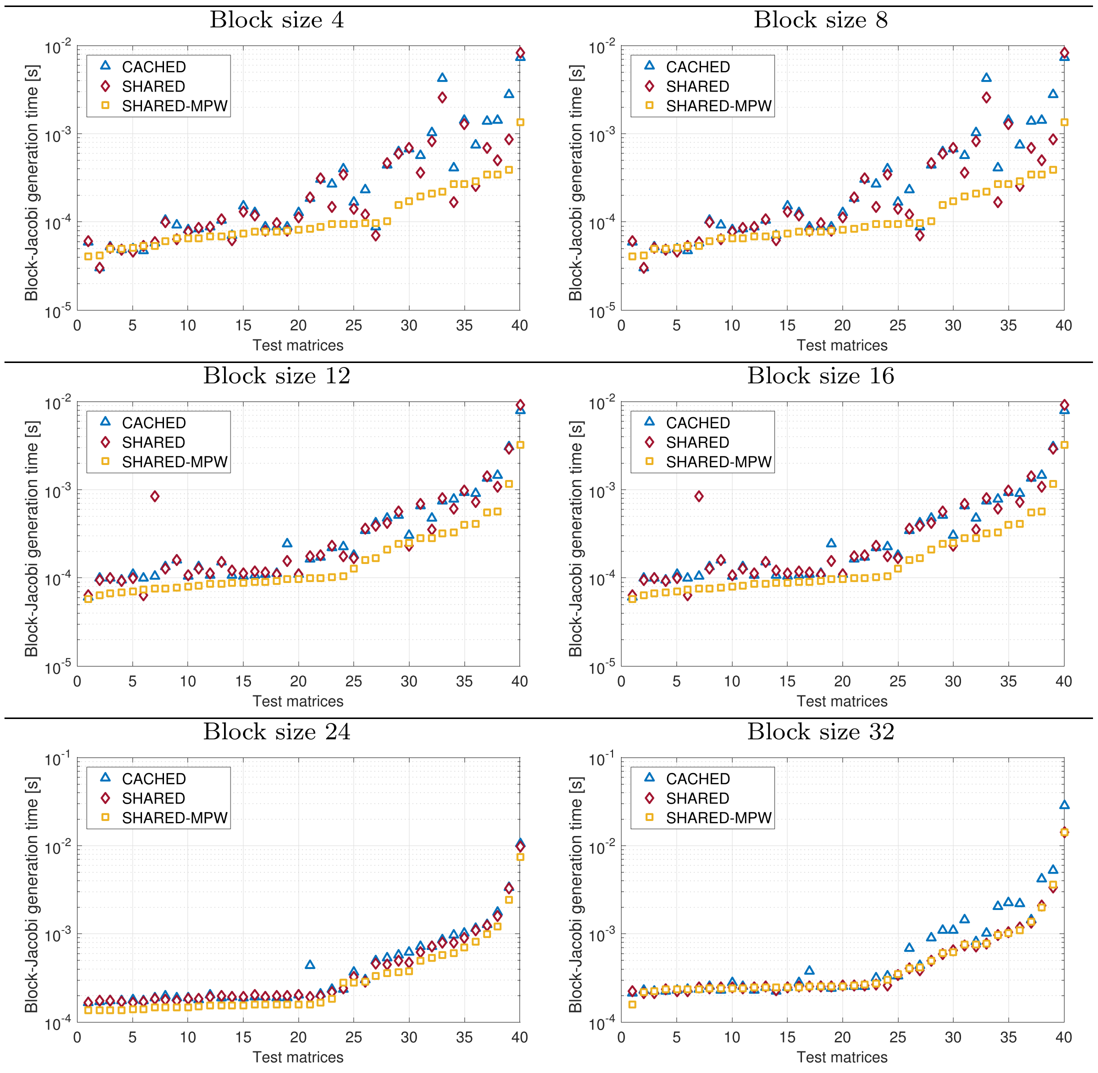

Fig. 10. Block-Jacobi generation time on NVIDIA P100 for a set of matrices taken from the SuiteSparse sparse matrix collection and varied block sizes: 4, 8, 12, 16, 24 and 32.

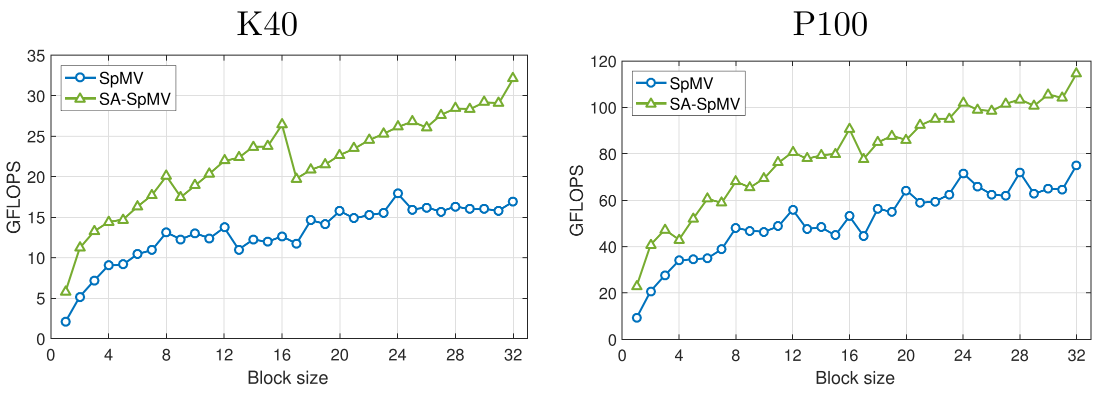

Fig. 11. Performance comparison of the block-Jacobi preconditioner application. SPMV is the generic sparse matrix-vector product routine from [2]. SA-SPMV is the specialized batched GEMV–based kernel developed as part of this work.

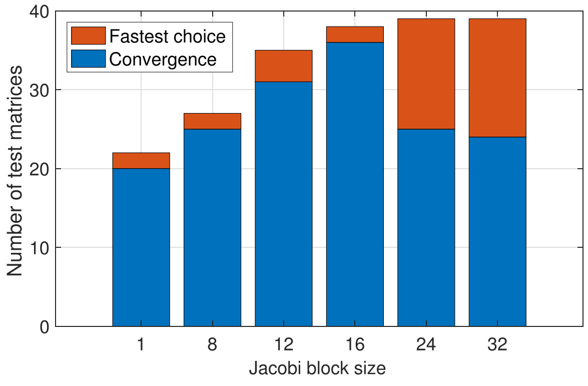

Fig. 12. IDR(4) convergence and performance comparison for different block sizes used in the block-Jacobi preconditioner. The problem matrices are listed along with key characteristics in Table 1.

4.3. Block-Jacobi generation

We now turn our attention to the complete block-inversion procedure that produces the block-Jacobi preconditioner. This includes the extraction of the diagonal blocks from a sparse data structure, followed by the explicit inversion, and then the insertion of the inverse blocks into the preconditioner matrix. As previously mentioned, the routines are "merged" into a single CUDA kernel that performs all three steps: (1) extraction from the sparse matrix structure, (2) inversion, and (3) insertion into the sparse preconditioner matrix. In this subsection, we compare three strategies for the generation of the block-Jacobi preconditioner, with the first strategy corresponding to an implementation that was already proposed in [2], and the last two strategies realizing the improvements described in Section 3:

1. CACHED: cached extraction/insertion with BGJE;
2. SHARED: shared extraction/insertion with BGJE; and
3. SHARED-MPW: shared extraction/insertion with BGJE-MPW.

Both SHARED and SHARED-MPW use the reduced memory shared extraction described in Section 3.2. Fig. 7 reveals that reducing the shared memory in the SHARED strategy can make the block-Jacobi generation up to four times faster. The problem-specific benefits depend on the upper bound for the block size, the pattern of the system matrix determining the actual size of the distinct diagonal blocks, and the hardware characteristics determining how many thread blocks a multiprocessor can schedule in parallel.

Because the performance of the extraction strategies depends on the structure of the problem matrix, we consider four nonzero distributions that are characteristic in sparse linear algebra. In Fig. 8, the arrow structure presents all nonzero entries on the (main) diagonal plus the last row/column of the matrix. In contrast, in the tridiagonal structure all nonzeros lie on the diagonal plus the diagonal immediately above/below it. These two structures are interesting, because they share the same nonzero count but exhibit different nonzero distributions. The other two examples correspond to a random block-diagonal matrix structure with nonzeros only in the diagonal blocks. The Laplace structure arises from the five-point stencil discretization of the Laplace equation.

In Fig. 9, we report the total execution time of the three block-Jacobi generation strategies applied to the four matrix structures. In this experiment, we fix the size of the matrix to 1,000,000 and increase the size of the diagonal blocks from 1 to 32.

For the arrow sparsity structure, the SHARED strategy is much faster than its CACHED counterpart; see results in the first row of Fig. 9. This result was expected because the arrow nonzero pattern contains a single dense row, which results in dramatic load imbalance if each row is traversed by a single thread, as is the case for CACHED. The SHARED alternative uses all threads of the warp to traverse this row, which alleviates the load imbalance and ensures coalescent access. For the other cases, the impact of non-coalescent memory access featured by CACHED is small as long as we consider small block sizes. This is because, for small blocks, only a few threads in each warp read data, which results in a reduced number of memory requests. Conversely, for large block sizes, the increase in memory requests impairs performance. Both strategies based on shared extraction eliminate load imbalance and non-coalescent memory access. Nonetheless, the reduced number of idle threads makes the SHARED-MPW version the overall winner.

We now asses the performance of the extraction routines for a set of test matrices from the SuiteSparse matrix collection. For brevity, we display the results for the P100 GPU only. The selected test matrices are listed along with some key properties in Table 1. In Fig. 10, we report the runtime of the block-Jacobi preconditioner generation for different block sizes. In these tests, the block sizes only correspond to an upper bound, and the blocks are identified via supervariable blocking. Also, some blocks can be smaller to better reflect the block structure of the problem matrix [11]. We again identify SHARED-MPW as the overall winner.

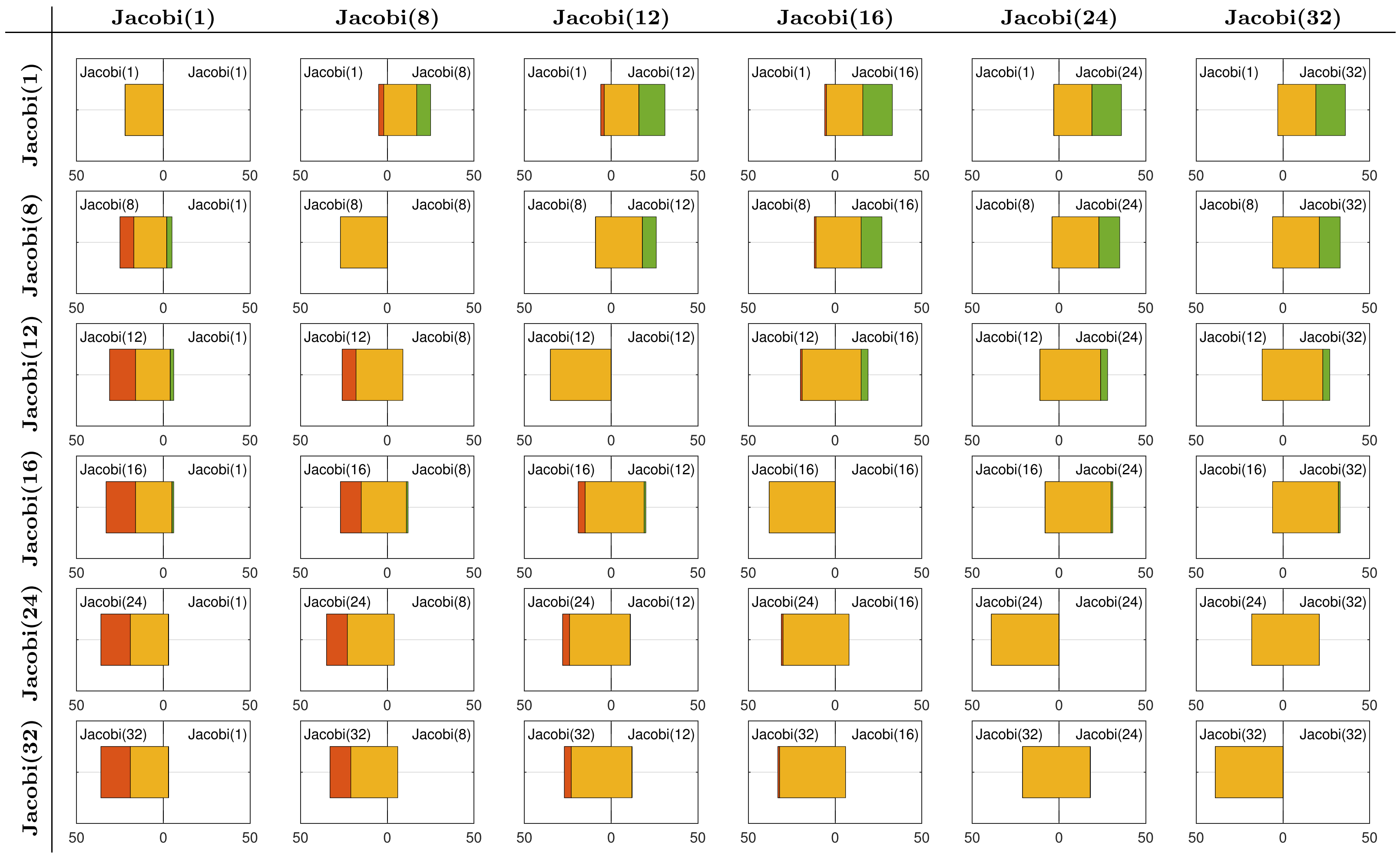

Fig. 13. Detailed comparison of IDR(4) enhanced with block Jacobi using different block sizes. (For interpretation of the references to color in this figure, the reader is referred to the web version of this article.)

4.4. Block-Jacobi application

In an iterative solver setting, the efficiency of a preconditioner depends on the overhead of generating the preconditioner and, to even a larger extent, on the cost of applying it during the iterative solution process.

In Fig. 11, we assess the performance of the preconditioner application using a generic SPMV kernel proposed in [2] versus our structure-aware SPMV (SA-SPMV) introduced in Section 3.3. On both architectures, SA-SPMV outperforms the initial SPMV kernel for the preconditioner application. For a block size of 32, this routine achieves about 32 gigaFLOPS on the K40 architecture, and around 80 gigaFLOPS on the P100 architecture. Local performance peaks can be identified for block sizes 8 and 16.

4.5. Convergence in the context of an iterative solver

Table 1 in the appendix details the convergence rate and execution time of an IDR(4) iterative solver [12] enhanced with either a scalar-Jacobi preconditioner or a block-Jacobi preconditioner for the selected cases from the SuiteSparse collection. The execution time includes both the preconditioner generation and the iterative solver execution. Detailed analysis reveals that in 88% of the tests, the preconditioner setup accounts for less than 1% of the total execution time. In all other cases, the block-inverse generation accounts for less than 5%. We combine the best kernels for the distinct preconditioner building blocks, i.e., `SHARED-MPW`, `BGJE-MPW`, and `SA-SPMV`. Other kernels are taken from the MAGMA-sparse open-source software package [10]. The IDR method [13] is among the most robust Krylov solvers [14]. The GPU-implementation of IDR available in MAGMA-sparse has been proven to achieve performance close to the hardware-imposed bounds [15]. For brevity, we run these tests on the newer P100 architecture only. We start the iterative solution process with an initial guess, $x_0 = 0$, solve for a right-hand side composed of random values in the interval [0, 1], and stop the iteration process once the relative residual norm has decreased by nine orders of magnitude. We allow for up to 50,000 iterations of the IDR(4) solver. In Fig. 12, we summarize the results showing for how many problems a certain configuration was the best choice (i.e., provided the fastest time to solution), and for how many problems a certain configuration was “successful” (concretely, reduced the relative residual norm by nine orders of magnitude within the limit of 50,000 iterations).

The results reveal that the scalar version of Jacobi fails to sufficiently improve the convergence of IDR(4) for a significant fraction of the test matrices. For the test matrices where IDR(4) preconditioned with the scalar Jacobi converges, the faster convergence obtained from using a block-Jacobi preconditioner typically compensates for the higher costs of preconditioner setup and application. In Fig. 13, we offer a head-to-head comparison of different block-size bounds for the block-Jacobi preconditioner used in IDR(4). The orange area in the plot at position “row Jacobi(x) vs. column Jacobi(y)” visualizes the number of matrices for which IDR(4) preconditioned with block-Jacobi of block size x converged, while it failed to converge with block size y. The opposite scenario, where block size y converged but block size x did not, is shown in green. Finally, the yellow area represents the number of matrices for which both methods converged—the area to the right of the center represents cases where block size y converges faster, while the area left of the center represents cases where block size x converges faster. The results suggest that adopting a larger block size usually leads to a more robust solver (i.e., convergence is achieved for a larger number of problems), and that a larger block size also improves the overall time-to-solution performance. However, in order to obtain the optimal performance for a specific problem, the block size should be tuned to the underlying block structure of the problem.

Overall, the results presented in this subsection offer strong evidence that the routines we developed provide an efficient approach to generating and applying a block-Jacobi preconditioner.

5. Concluding remarks

In this paper, we presented an enhanced, variable-size batched matrix inversion routine for GPUs based on the GJE process. Our approach replaces explicit pivoting with a strategy that reassigns the workload instead of shuffling the data and relies heavily on CUDA’s latest warp-local communication features. As a result, our matrix inversion kernel is more flexible and significantly outperforms its counterparts in the cuBLAS library. In the framework of block-Jacobi preconditioning, we combined the batched matrix inversion procedure with efficient routines for extracting the diagonal blocks from the sparse data structures (where the problem matrix is stored) and inserting the inverse blocks back into the preconditioner. We also addressed the efficient preconditioner application by developing a structure-aware batched kernel for the sparse matrix-vector product that accommodates variable-size matrix operands. Finally, we demonstrated that block Jacobi can be significantly more efficient than a scalar Jacobi when preconditioning iterative solvers.

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. H. Anzt was supported by the “Impuls und Vernetzungsfond of the Helmholtz Association” under grant VH-NG-1241. G. Flegar and E. S. Quintana-Ortí were supported by project TIN2014-53495-R of the MINECO–FEDER; and project OPRECOMP (http://oprecomp.eu) with the financial support of the Future and Emerging Technologies (FET) programme within the European Union's Horizon 2020 research and innovation programme, under grant agreement No 732631. The authors would also like to acknowledge the Swiss National Computing Centre (CSCS) for granting computing resources in the Small Development Project entitled "Energy-Efficient preconditioning for iterative linear solvers" (#d65).

References

[1] Y. Saad, Iterative Methods for Sparse Linear Systems, second ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.
[2] H. Anzt, J. Dongarra, G. Flegar, E.S. Quintana-Ortí, Batched Gauss-Jordan Elimination for Block-Jacobi Preconditioner Generation on GPUs, in: Proceedings of the Eighth International Workshop on Programming Models and Applications for Multicores and Manycores, PMAM'17, ACM, New York, NY, USA, 2017, pp. 1–10, doi:10.1145/3026937.3026940.
[3] A.S. Householder, The Theory of Matrices in Numerical Analysis, Dover, New York, 1964.
[4] E.S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, R. van de Geijn, A note on parallel matrix inversion, SIAM J. Sci. Comput. 22 (5) (2001) 1762–1771.
[5] P. Benner, P. Ezzatti, E. Quintana-Ortí, A. Remón, Matrix inversion on CPU-GPU platforms with applications in control theory, Concurr. Comput. Pract. Exp. 25 (8) (2013) 1170–1182.
[6] J. Kurzak, H. Anzt, M. Gates, J. Dongarra, Implementation and tuning of batched Cholesky factorization and solve for NVIDIA GPUs, IEEE Trans. Parallel Distrib. Syst. 27 (7) (2016) 2036–2048, doi:10.1109/TPDS.2015.2481890.
[7] A. Haidar, T. Dong, P. Luszczek, S. Tomov, J. Dongarra, Batched matrix computations on hardware accelerators based on GPUs, Int. J. High Perform. Comput. Appl. 29 (2) (2015) 193–208.
[8] J. Dongarra, I.S. Duff, M. Gates, A. Haidar, S. Hammerling, J. Higham, J. Hogg, P. Valero-Lara, D. Relton, S. Tomov, M. Zounon, A Proposed API for Batched Basic Linear Algebra Subprograms, The University of Manchester, 2016, pp. 1749–9097. ISSN
[9] H. Anzt, E. Chow, T. Huckle, J. Dongarra, Batched generation of incomplete sparse approximate inverses on GPUs, in: Proceedings of the Seventh Workshop on Scalable Algorithms for Large-scale Systems, ScalA'16, 2016.
[10] Innovative Computing Lab, Software Distribution of MAGMA Version 2.0, 2016. http://icl.cs.utk.edu/magma/.
[11] E. Chow, H. Anzt, J. Scott, J. Dongarra, Experimental Study of Iterative Methods and Blocking for Solving Sparse Triangular Systems in Incomplete Factorization Preconditioning, Journal of Parallel and Distributed Computing, submitted.
[12] 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.
[13] 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 Second 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.
[14] H. Anzt, J. Dongarra, M. Kreutzer, G. Wellein, M. Koehler, Efficiency of general Krylov methods on GPUs – an experimental study, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2016, pp. 683–691.
[15] 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. Appl. (2016), doi:10.1177/1094342016646844.