IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received July 25, 2018, accepted September 6, 2018, date of publication September 19, 2018,
date of current version December 7, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2871219

A GPU Solver for Sparse Generalized Eigenvalue Problems With Symmetric Complex-Valued Matrices Obtained Using Higher-Order FEM

ADAM DZIEKONSKI AND MICHAL MROZOWSKI, (Fellow, IEEE)
Faculty of Electronics, Telecommunications, and Informatics, Gdańsk University of Technology, 80-233 Gdańsk, Poland

Corresponding author: Michal Mrozowski (m.mrozowski@ieee.org)

This work was supported in part by the Polish National Science Centre under Contract DEC-2014/13/B/ST7/01173, in part by the Foundation for Polish Science within the TEAM-TECH Programme, through the European Regional Development Fund, Smart Growth Operational Programme 2014–2020 (Project EDISOn: Electromagnetic Design of flexIble SensOrs), and in part by the statutory funds from the Faculty of Electronics, Telecommunications, and Informatics, Gdańsk University of Technology.

ABSTRACT This paper discusses a fast implementation of the stabilized locally optimal block preconditioned conjugate gradient method, using a hierarchical multilevel preconditioner to solve non-Hermitian sparse generalized eigenvalue problems with large symmetric complex-valued matrices obtained using the higher-order finite-element method (FEM), applied to the analysis of a microwave resonator. The resonant frequencies of the low-order modes are the eigenvalues of the smallest real part of a complex symmetric (though non-Hermitian) matrix pencil. These types of pencils arise in the FEM analysis of resonant cavities loaded with a lossy material. To accelerate the computations, graphics processing units (GPU, NVIDIA Pascal P100) were used. Single and dual-GPU variants are considered and a GPU-memory-saving implementation is proposed. An efficient sliced ELLR-T sparse matrix storage format was used and operations were performed on blocks of vectors for best performance on a GPU. As a result, significant speedups (exceeding a factor of six in some computational scenarios) were achieved over the reference parallel implementation using a multicore central processing unit (CPU, Intel Xeon E5-2680 v3, and 12 cores). These results indicate that the solution of generalized eigenproblems needs much more GPU memory than iterative techniques when solving a sparse system of equations, and also requires a second GPU to store some data structures in order to reduce the footprint, even for a moderately large systems.

INDEX TERMS Generalized eigenvalue problem, FEM, complex-valued sparse matrix pencil, GPU, Maxwell's equations.

I. INTRODUCTION

Graphics processing units offer impressive performance that has been demonstrated in solving many linear algebra problems, including finding the eigenvalues and eigenvectors of real- and complex-valued matrices. However, remarkable speed gains over CPU-based solutions have mainly been reported in dense cases. Sparse eigenvalue problems require iterative algorithms, and the performance of sparse solvers is limited by the efficiency of matrix–vector multiplication. This operation is memory-bandwidth bound, so to maximize performance, data structures need to be optimized for the GPU-architecture, while transfers to and from memory should be minimized. This implies that the algorithms for solving sparse problems should be redesigned to make best use of the fast on-board GPU memory.

II. Related Work

While iterative techniques for solving a system of sparse equations on one or many GPUs have been considered in numerous publications [1]–[9], the literature on solving sparse eigenvalue problems on a GPU is scarce; only algorithms for real and complex Hermitian matrices have been considered [10]–[15]. The focus is on standard eigenvalue problems of the form $\mathbb{K}\mathbf{x} = \sigma\mathbf{x}$. In practice, one often needs to investigate the spectrum of a sparse matrix pencil, which means that one or more eigenvalues of the generalized problem $\mathbb{K}\mathbf{x} = \sigma\mathbb{M}\mathbf{x}$ are of interest. For instance, generalized eigenvalue problems arise in modal analysis using the finite-element method, which is one of the most important numerical techniques for solving partial differential equations. In most cases, a few of the smallest eigenvalues are to be

2169-3536 © 2018 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

found. These eigenvalues correspond to the free oscillations of a system. If there is no loss in the system, the eigenvalues are real, and both matrices in the pencil are symmetric and real, with $\mathbb{M}$ being additionally positive definite. In the presence of loss, the matrices are symmetric but no longer real—at least one of them must be complex-valued. Consequently, the eigenvalues are also complex, with the imaginary part corresponding to damping. Computation of damped and undamped resonances by means of the finite-element methods is important for acoustics, electromagnetics, and modal analysis in structural dynamics.

From this description, it is evident that GPU-accelerated solvers that are capable of dealing with symmetric complex-valued non-Hermitian matrices are also needed for sparse generalized eigenvalue problems. It should be noted that, unlike in standard eigenvalue problems, two matrices need to be stored and processed by the solver. This means that more GPU memory is used in these computations than in solving a standard problem. When the matrices are complex, the memory requirements double. The amount of fast memory on a single GPU is limited to 12–16 GB, and it is important to take this into account when implementing a particular algorithm. Otherwise, the size of the matrices that the GPU solver can handle may be so small that it might actually be faster to solve the problem on a CPU. To make use of GPU computational power for larger generalized problems (especially when the matrices are complex), it may be necessary to use more than one accelerator and to distribute the matrices, and possibly also the other data structures, between different GPUs.

In this paper, we present a GPU implementation of the recently developed stabilized locally optimal block preconditioned conjugate gradient (LOBPCG) procedure for solving large sparse generalized symmetric eigenvalue problems arising in computational electromagnetics in the finite-element analysis of lossy resonators. It should be noted that LOBPCG is intended for finding the smallest eigenvalues of problems with real matrices [16], and this algorithm would normally have been regarded as unsuitable for treating problems with complex-valued matrices. Recently, however, we have demonstrated [17] that, with a small modification, the LOBPCG method can be successfully applied to the analysis of resonates with small and moderate loss. In the present work, we use this modified LOBPCG algorithm and present its implementation and performance on a low-cost computational server with a GPU accelerator. To handle larger problems, we propose a GPU-memory-saving implementation and consider a setup involving two GPUs, each sporting 12 GB of fast RAM (random access memory). This paper extends our recent work on the GPU-accelerated LOBPCG algorithm for generalized eigenvalue problems [18] to a new algorithm: the stabilized LOBPCG method for the sparse generalized eigenvalue problem with complex-valued matrices, and it discusses its single-GPU and dual-GPU implementations. To improve convergence, we propose a new preconditioner that is geared towards the higher-order finite element method. This preconditioner takes advantage of the hierarchy of the basis functions. The generalized eigenvalue problem we solve we has large nullspace that is spanned by spurious, nonphysical eigenvectors. To filter out this nullspace, we use the preconditioned conjugate gradient method. The sLOBPCG algorithm is more demanding of memory than the iterative algorithms for sparse linear algebra considered to date. We thus propose a new GPU-memory-saving implementation, in which we decompose the pair of matrices in the matrix pencil $(\mathbb{K}, \mathbb{M})$, into blocks. These blocks, rather than the entire matrices, are then used when applying the preconditioner, and are also employed in sLOBPCG iterations when constructing the projection subspace (computing residuals, and projected pencil matrices for the Rayleigh–Ritz method). To this end, we also consider a dual-GPU implementation in which the constituent blocks of pencil matrices are split between two accelerators in such a way that transfer between the two GPUs is minimized and load balancing is achieved while the sparse matrix–vector product is computed. We also obtain increased computational intensity by vector blocking.

The paper is organized as follows: Section II briefly describes the finite element formulations used to compute the resonant frequencies of a microwave cavity loaded with a lossy dielectric material. Section III discusses the LOBPCG algorithm, as modified for complex-valued problems, and the preconditioner and memory requirements. The implementation of our approach—including the choice of a matrix format for the sparse matrix–vector (SpMV) and sparse matrix–matrix (SpMM) product—is described in Section IV. Finally, the results and a comparison with the CPU-implementation optimized for multicore architectures are presented and discussed in Section V for various computational scenarios.

III. GENERALIZED EIGENVALUE PROBLEM WITH COMPLEX-VALUED MATRICES

We consider a resonant electromagnetic cavity Ω enclosed by a boundary S (a perfect electric conductor) and loaded with a dielectric material. The unforced oscillations in this cavity are governed by the vector Helmholtz equation:

$$\nabla \times \left(\frac{1}{\mu_r} \nabla \times \vec{E} \right) - k_0^2 \epsilon_r \vec{E} = 0, \tag{1}$$

where $\vec{E}$ is the electric field, $k_0 = \omega / c$ is the wavenumber (ω being the angular frequency and c the speed of light), and ϵ_r and μ_r are the relative permittivity and permeability, respectively.

In order to find the resonances of the cavity, we apply a 3D vector finite element method formulation with hierarchical vector basis functions up to the third order [19]. Following the standard FEM procedure [20], we obtain:

$$(\mathbb{K} - \sigma \mathbb{M}) \mathbf{e} = 0, \tag{2}$$

where $\sigma = \left(\frac{\omega}{c}\right)^2$, $\mathbb{K}, \mathbb{M} \in \mathbb{C}^{n \times n}$ are symmetric sparse stiffness and mass matrices, respectively, and n is the number of degrees of freedom (DoF). If no energy dissipation occurs, both matrices are real-valued and $\mathbb{M}$ is positive definite, so all eigenvalues σ are positive and real.

MOST WIEDZY Downloaded from mostwiedzy.pl

If we now assume that the dielectric is lossy and its material properties are described by a complex-valued frequency-independent permittivity, $\mathbb{M}$ becomes non-Hermitian complex symmetric ($\mathbb{M} = \mathbb{M}^T$, $\mathbb{M} \neq \mathbb{M}^H$). The consequence of this is that the spectrum is complex. The real part of σ corresponds to the free oscillation frequency, while the imaginary part determines the quality factor given by Eq. 8. To find a few low-order resonances of the cavity, we have to solve a generalized eigenproblem (2) for several nonzero eigenvalues with the smallest real part. This can be achieved by using the shift-and-invert technique [21], which involves factorizing a large sparse matrix, which is time-consuming and has high memory requirements. As the number of degrees of freedom grows, the number of nonzero elements in the factors increases rapidly, and the shift-and-invert technique becomes impossible because of the limitations of CPU memory.

IV. STABILIZED LOBPCG ALGORITHM FOR COMPLEX SYMMETRIC GEP

One of the most efficient and most rapidly converging iterative algorithms that can be used to compute small nonzero eigenvalues of a generalized eigenvalue problems, and which does not involve large matrix factorization, is the locally optimal block preconditioned conjugate gradient (LOBPCG) method. To prevent this method from converging to zero eigenvalues associated with the nullspace of the discretized curl–curl operator, the nullspace must be filtered out. Since the zero eigenvalues of the discretized curl–curl operator have nonvanishing divergence, filtering can be achieved by explicitly enforcing the divergence-free condition during iterations, as demonstrated in [22].

The LOBPCG algorithm also requires that both matrices $\mathbb{K}$ and $\mathbb{M}$ be real or complex-Hermitian at most. For lossy systems, the Hermitian symmetry required by LOBPCG does not hold. As shown in [17] the original LOBPCG algorithm does not converge for cavities loaded with even a sightly lossy dielectric. The lack of convergence is due to the inadequate selection of the projection basis in one of the algorithm's steps. A simple algorithmic modifications has been proposed to stabilize convergence of the LOBPCG algorithm [17]. The modification involves sorting the Ritz values in each iteration and retaining only those that are close to the target value. This enhancement stabilized the convergence of LOBPCG when solving eigenproblems associated with lossy electromagnetic systems. Algorithm 1 shows the main steps in the sLOBPCG method. In general, the algorithm looks for Ritz values and Ritz vectors in the trial space consisting of the current eigenvector approximation, the preconditioned residual, and the implicitly computed difference between the current and the previous eigenvector approximations [16]. The projection of the original problem onto the trial space is carried out in Line 15. Preconditioned residuals are computed in lines 7 and 11. Lines 8 and 9 are a convergence check. The converged eigenvectors are removed from the trial space. Lines 1 and 12 implement nullspace filtering. Almost all steps

Algorithm 1 Stabilized LOBPCG for Real or Complex Symmetric Problems. Inputs: $\mathbb{K}$, $\mathbb{M}$ Are Sparse Real or Complex Symmetric $n \times n$ Matrices, $\mathscr{P}$ Is a Preconditioner, $\mathbf{Y}$ Is a Basis in the Nullspace, $\mathbf{X}_0$ Is the Initial Block Vector of Size $n \times (q+1)$, q Is the Number of Eigenvalues to be Computed, and MaxIter Is the Maximum Number of Iterations. γ Is the Shift Value Used in the Sorting Algorithm

The outputs of the sLOBPCG method are the q smallest-magnitude nonzero eigenvalues $\{\sigma_1, \ldots, \sigma_q\}$ stored in a diagonal matrix Σ_{output} and a dense block vector $\mathbf{X}_{output}$ of size $n \times q$, which consists of the q respective eigenvectors.

0. Initialize $\tilde{\mathbf{P}}_0 = \mathbf{P}_0 = []$
1. $\mathbf{X}_0 \leftarrow \mathbf{X}_0 - \mathbf{Y}\,(\mathbf{Y}^T\mathbb{M}\mathbf{Y})^{-1}((\mathbb{M}\mathbf{Y})^T\mathbf{X}_0)$ ▷ Make $\mathbf{X}_0$ $\mathbb{M}$-orthogonal to the nullspace
2. $\mathbb{M}$-orthogonalize columns of $\mathbf{X}_0$
3. Compute $(\mathbf{X}_0^T\mathbb{K}\mathbf{X}_0)\tilde{\mathbf{S}}_0 = \tilde{\mathbf{S}}_0\tilde{\Sigma}_0$, where $\tilde{\Sigma}_0 = diag(\sigma_1, \ldots, \sigma_{q+1})$ ▷ Spectral decomposition
4. $(\Sigma_0, \mathbf{S}_0) \leftarrow$ Eigenpairs $(\sigma_i, \tilde{\mathbf{S}}_0\mathbf{e}_i)$ sorted by $|\sigma_i - \gamma|$ in the ascending order.
5. $\mathbf{X}_0 \leftarrow \mathbf{X}_0\mathbf{S}_0$
6. **for** $k = 0 : (\text{MaxIter} - 1)$ **do**
7. Compute residuals $\tilde{\mathbf{R}}_k = \mathbb{K}\mathbf{X}_k - \mathbb{M}\mathbf{X}_k\Sigma_k$
8. Find $D = \left\{i : \|\tilde{\mathbf{R}}_k\mathbf{e}_i\|_2 > \epsilon\right\}$, $\tilde{q}$ = size of D
9. **if** ($\tilde{q} = 1$ **AND** i=q+1) **OR** ($\tilde{q} = 0$) **then** ▷ Convergence check
   exit
   end if
10. Let $\mathbf{R}_k = [\tilde{\mathbf{R}}_k(:, j)]_{j\in D}$
11. Apply preconditioner $\mathbf{H}_k = \mathscr{P}^{-1}\mathbf{R}_k$
12. $\mathbf{H}_k \leftarrow (\mathbf{I} - \mathbf{Y}(\mathbf{Y}^T\mathbb{M}\mathbf{Y})^{-1}((\mathbb{M}\mathbf{Y})^T)\mathbf{H}_k$ ▷ Filtering out nullspace components
13. $\mathbf{H}_k \leftarrow \mathbf{H}_k - \mathbf{X}_k((\mathbb{M}\mathbf{X}_k)^T\mathbf{H}_k)$▷ Orthogonalization vs. eigenvector approximations
14. If $\tilde{\mathbf{P}}_k$ nonempty, set $\mathbf{P}_k = [\tilde{\mathbf{P}}_k(:, j)]_{j\in D}$
15. Compute $\tilde{\mathbf{K}} = [\mathbf{X}_k,\mathbf{H}_k,\mathbf{P}_k]^T\mathbb{K}[\mathbf{X}_k, \mathbf{H}_k, \mathbf{P}_k]$, $\tilde{\mathbf{M}} = [\mathbf{X}_k,\mathbf{H}_k,\mathbf{P}_k]^T\mathbb{M}[\mathbf{X}_k, \mathbf{H}_k, \mathbf{P}_k]$
16. Compute $\tilde{\mathbf{K}}\tilde{\mathbf{S}}_k = \tilde{\mathbf{M}}\tilde{\mathbf{S}}_k\tilde{\Sigma}_k$, where $\tilde{\Sigma}_k = diag(\sigma_1, \ldots, \sigma_{(q+1)+2\tilde{q}})$ ▷ Spectral decomposition
17. $(\Sigma_k, \tilde{\mathbf{S}}_k)\leftarrow$ Eigenpairs $(\sigma_i, \tilde{\mathbf{S}}_k\mathbf{e}_i)$ sorted by $|\sigma_i - \gamma|$ in the ascending order.
18. $\mathbf{S}_k = \tilde{\mathbf{S}}_k[\mathbf{e}_1, \ldots, \mathbf{e}_{q+1}]$, $\Sigma = \Sigma_k(1 : q+1, 1 : q+1)$
19. $\tilde{\mathbf{P}}_k = [\mathbf{R}_k\mathbf{P}_k]\mathbf{S}_k(q+1 : (q+1+2\tilde{q}), :)$
20. $\mathbf{X}_k \leftarrow \mathbf{X}_k\mathbf{S}_k(1 : q+1, :) + \tilde{\mathbf{P}}_k$
21. **end for**
22. Eigenpairs: $\mathbf{X}_{output} = \mathbf{X}(1 : n, 1 : q)$ and $\Sigma_{output} = \Sigma(1 : q, 1 : q)$.

are identical to those of the original LOBPCG algorithm [16], [22] - the only part which is different are lines 4 and 17 which perform the sorting that stabilizes the algorithm, given γ as a target: a rough approximation to the smallest eigenvalue (this may be known or found by solving a smaller problem with the shift-and-invert algorithm). The nullspace of the curl–curl operators is spanned by the eigenvectors x corresponding to

MOST WIEDZY Downloaded from mostwiedzy.pl

Algorithm 2 Hierarchical Multilevel Preconditioner (*Hie-ML*)

1. $\mathbf{z} = \text{Hie-ML}(\mathbf{r},\text{i})$
2. $\mathbf{z} = 0$
3. if i == 1 then
4. $\mathbf{z} = \mathbb{A}_{11}^{-1}\mathbf{r}$ // solve the lowest level
5. else
6. smoothing($\mathbf{z}$, $\mathbf{r}$) // the highest level
7. $\mathbf{r}^{i-1} = \mathbf{r} - \mathbb{A}_{i-1,i}\,\mathbf{z}$
8. $\mathbf{z}^{i-1} = \text{Hie-ML}(\mathbf{r},\text{i-1})$
9. $\mathbf{r}^{i} = \mathbf{r} - \mathbb{A}_{i,i-1}\,\mathbf{z}^{i-1}$
10. smoothing($\mathbf{z}$, $\mathbf{r}$) // the highest level

eigenvalues equal to zero that satisfy $\mathbb{Y}^T\mathbb{M}x \neq 0$, where $\mathbb{Y}$ is a gradient projection matrix, which means that these are fields that do not have zero divergence ($\nabla \cdot (\epsilon \mathbf{E}) \neq 0$). These solutions are not physical and should be eliminated—otherwise the sLOBPCG would converge to these spurious eigensolutions rather than to physical ones. The elimination of spurious solutions from the projection subspace is carried out in each iteration of sLOBPCG and involves the solution of a linear system with a matrix ($\mathbb{A} = \mathbb{Y}^T\mathbb{M}\mathbb{Y}$). The matrix $\mathbb{Y}$ is very sparse, with at most two nonzero entries with values of 1 or -1 in each row [22].

A. PRECONDITIONER

The convergence of LOBPCG depends on the choice of the application-specific preconditioner $\mathscr{P}$. Since the FEM formulation we consider in this paper involves higher-order basis functions, we use a multilevel preconditioner that takes advantage of the hierarchy of basis functions applied in the FEM method. We used this preconditioner in our previous work to solve real and complex systems of equations [3], [4], [9] and also for preconditioning real eigenproblems solved with classical LOBPCG [18]. The number of levels depends on the order of the basis functions. The FEM code we used [23] allowed for basis functions of up to the third order (QTCuN) [19] so, in our case, the number of levels is three. In a hierarchical multilevel preconditioner, a global sparse matrix $\mathbb{A}$ is divided in submatrices ($\mathbb{A}_{ij}$) that are related to the orders of the finite element basis functions. The matrix used for preconditioning was formed from the stiffness and mass matrices as $\mathbb{A} = \mathbb{K} - \kappa^2\mathbb{M}$, where κ is a constant. The value of κ should be close to the eigenvalues that are being sought. The division is as follows:

$$\begin{bmatrix} \mathbb{A}_{11} & \mathbb{A}_{12} & \mathbb{A}_{13} \\ \mathbb{A}_{21} & \mathbb{A}_{22} & \mathbb{A}_{23} \\ \mathbb{A}_{31} & \mathbb{A}_{32} & \mathbb{A}_{33} \end{bmatrix} = \begin{bmatrix} \mathbb{K}_{11} & \mathbb{K}_{12} & \mathbb{K}_{13} \\ \mathbb{K}_{21} & \mathbb{K}_{22} & \mathbb{K}_{23} \\ \mathbb{K}_{31} & \mathbb{K}_{32} & \mathbb{K}_{33} \end{bmatrix} - \kappa^2 \begin{bmatrix} \mathbb{M}_{11} & \mathbb{M}_{12} & \mathbb{M}_{13} \\ \mathbb{M}_{21} & \mathbb{M}_{22} & \mathbb{M}_{23} \\ \mathbb{M}_{31} & \mathbb{M}_{32} & \mathbb{M}_{33} \end{bmatrix}$$

Normally, just one V-cycle of the hierarchical multilevel preconditioner is enough for convergence. One V-cycle was used in our previous work on the GPU-accelerated LOBPCG method for real valued generalized eigenvalue problems [18]. However, trials of the stabilized version of LOBPCG intended for complex GEPs revealed that a single cycle is not sufficient. In the sLOBPCG method presented in Algorithm 1, this was therefore replaced by iterations of the preconditioned conjugate gradient solver (PCG-V) [3], [4], [9], which also executes one V-cycle of the hierarchical multilevel preconditioner for each conjugate–gradient iteration (Algorithm 2). The number of iterations of the PC-V method depends on the loss, and ranges from a few for lossless and weakly lossy structures to a few tens for larger losses.

TABLE 1. Sparse and dense matrices used in sLOBPCG. $\mathscr{P}$: PCG-V solver; NSF: nullspace filtering; q: number of eigenvalues to be computed; k: order of the nullspace.

Description	Matrix name	Size	Type
Stiffness	$\mathbb{K}$	$n \times n$	sparse (real for problem (1)
Mass	$\mathbb{M}$	$n \times n$	sparse (complex)
$\mathscr{P}$	$\mathbb{A} = \mathbb{K} - \kappa^2\mathbb{M}$ PCG-V vectors	$n \times n$ $n \times (q+1)$	sparse (complex) dense (complex)
NSF	$\mathbb{A}_{NSF} = \mathbb{G}^T\,\mathbb{M}\,\mathbb{G}$ PCG-V vectors	$k \times k$ $k \times (q+1)$	sparse (complex) dense (complex)
LOBPCG	$\mathbf{P},\mathbf{R},\mathbf{X}$ $\mathbf{KP},\mathbf{KR},\mathbf{KX}$ $\mathbf{MP},\mathbf{MR},\mathbf{MX}$ $\mathbb{G}$ $\mathbb{MG} = \mathbb{M}\,\mathbb{G}$	$n \times (q+1)$ $n \times (q+1)$ $n \times (q+1)$ $n \times k$ $n \times k$	dense (complex) dense (complex) dense (complex) sparse (real) sparse (complex)

B. MEMORY REQUIREMENTS

In order to obtain the highest performance, all data should reside in the GPU memory. The memory requirements of sLOBPCG are higher than for other iterative algorithms considered in the literature on GPU-accelerated sparse linear algebra. To substantiate this claim, the data structures required to implement the sLOBPCG method (Algorithm 1) are shown in Table 1. The stiffness matrix can in general be complex. However, for the type of resonant cavities we consider in our work, it is always real.

It can be seen that, in addition to sparse matrices (mass, stiffness, preconditioner, and nullspace), there are a number of dense tall and skinny complex-valued matrices. The number of columns depends on the number of eigenvalues q that we seek.

V. IMPLEMENTATION

The amount of GPU memory available is limited and the sLOBPCG algorithm is more demanding of memory than the iterative algorithms for sparse linear algebra considered to date. Consequently, these methods have to be implemented with this in mind. To save GPU memory, the precondition matrix was constructed on-the-fly from the blocks of the stiffness and mass matrices. That is, instead of storing the entire matrix $\mathbb{A}$, we store 18 separate submatrices $\mathbb{K}_{ij}$, $\mathbb{M}_{ij}$, $i = 1, 3, j = 1, 3$. Also, when the problem is lossy, only the mass matrix is complex, so by decomposing the preconditioner, we can save additional memory by storing blocks of $\mathbb{K}_{ij}$, $i = 1, 3, j = 1, 3$ as sparse real-valued matrices

MOST WIEDZY Downloaded from mostwiedzy.pl

and blocks $\mathbb{M}_{ij}$, $i = 1, 3, j = 1, 3$ as sparse complex-valued matrices. The preconditioner involves directly solving the sparse system of linear equations on the lowest level. This operation is performed on a CPU using Intel's Pardiso direct solver. The remaining operations of smoothing and sparse matrix–vector product to transfer solutions between levels are carried out on a GPU. For smoothing, we perform a few iterations of a weighted Jacobi process. Because the blocks of the preconditioner matrix $\mathbb{A}$, are not stored explicitly, the smoothing operation and transfers between levels are implemented in this GPU-memory-saving variant as a sequence of matrix–vector multiply–add operations involving the relevant blocks of the stiffness and mass matrices and κ. In this respect, the implementation of the preconditioner differs from those presented in our earlier work [3], [4], [9], [18]. The computational kernels for GPU execution were programmed in CUDA and we used cuSPARSE (v8.0) for all the basic operations, except the sparse matrix–vector product.

A. SPARSE MATRIX-VECTOR AND SPARSE MATRIX-DENSE MATRIX MULTIPLICATION

Sparse matrix–vector multiplication is the cornerstone of the sLOBPCG algorithm. The performance of this operation on a GPU depends on the format used to represent sparse matrices [24]. It is known that the main architectural bottleneck to the efficiency of the SpMV algorithm is memory bandwidth. Performance can be improved by vector blocking [9], [11], [13], [25], [26]. We use blocking in our implementation, since, when the number of eigenvalues sought is $q > 1$, the sequence of matrix–vector multiplications needed to apply the preconditioner to multiple vectors, to solve a system of equations with multiple right-hand sides for nullspace filtering (steps 1,12), to compute residuals (step 7), to carry out orthogonalization (step 13), or to carry out the projection (step 15) can replaced by multiplications of a sparse matrix by a block of vectors. Therefore, it is also important to develop optimized kernels for multiplying sparse matrices by dense matrices, where the dense matrix is tall and skinny with the number of columns comparable to q (the number of columns decreases as the iteration progresses and subsequent eigenvalues converge).

B. PERFORMANCE BOUNDS

To determine the performance bounds for the SpMV (sparse matrix–vector product) and SpMM (sparse matrix–dense matrix product) kernels for both real and complex arithmetic, we use the approach presented in [11]. The upper bound is given by

$$P_{max} = \frac{Flops}{t_B} BW, \tag{3}$$

where t_B is lower bound on data transfers in bytes, $Flops$ is the number of floating point operations for an SpMV, and BW is the achievable data transfer rate to/from main memory

If the tall and skinny matrix is composed of m vectors of n elements each, and a sparse $n \times n$ matrix has n_z nonzero elements, the lower bound on data transfers in bytes and Flops is calculated as follows:

$$t_B = K_1 \cdot 8 \cdot n_z + 4 \cdot n_z + 4 \cdot \frac{n_z}{K_3} + K_1 \cdot 8 \cdot \frac{n_z}{K_3} \cdot m + K_1 \cdot 8 \cdot \frac{n_z}{K_3} \cdot m \tag{4}$$

$$Flops = K_2 \cdot 2 \cdot n_z m, \tag{5}$$

where $K_1 = 1, K_2 = 1$ for double precision, and $K_1 = 2$, $K_2 = 4$ for complex double precision, respectively; K_3 is the average number of nonzero elements per row. We consider factors related to n, since they become significant in t_B as the number of vectors (m) in the blocked sparse matrix vector product increases. The bandwidth for the hardware can be measured: for the NVIDIA K40 accelerator used in [11], the measured bandwidth, as reported by NVIDIA's `bandwidthTest` benchmark, is 180 GB/s, while for the NVIDIA P100 GPU this is 390 GB/s. The key from the performance of the PCG-V solver sparse matrix is $\mathbb{A}_{22} = \mathbb{K}_{22} - \kappa\mathbb{M}_{22}$, used on the highest level of the multilevel preconditioner from Algorithm 2. For this matrix $K_3 = 40$, and the upper bound for the SpMV and SpMM performance on a K40 computed using the above formula is 28.8 GFlop/s (m=1), 310.5 (m=16) and 68.9 GFlop/s (m=1), 698.2 GFlop/s (m=16) for real double precision and complex double precision, respectively. For the P100 accelerator, the upper limits are 62 GFlop/s (m=1) or 673 GFlop/s (m=16) for real double precision, and 149 GFlop/s (m=1) or 1513 GFlop/s (m=16) for complex double precision.

C. SPARSE MATRIX STORAGE FORMAT

The actual performance depends on the matrix sparsity pattern and the format used to store the sparse matrix [24]. In [11], the SELL-P sparse matrix storage format [25], [27] was used. In this work, we also consider Sliced ELLR-T (SELLR-T) [28]. Both SELL-P and SELLR-T formats take advantage of dividing the sparse matrix into slices/blocks of rows in order to reduce the memory overhead, and they employ multiple threads to perform computations in each row. In SELL-P, each slice/block is converted into ELLPACK format with the row-length of each block padded up to a multiple of the number of threads assigned to each row. In SELLR-T, a sparse matrix is reordered from least to the most populated row and then sliced (Figure 1). In each slice, a permutation based on ELLR-T proposed in [29] is utilized in the preprocessing stage. The rows are padded with zeros, so that the number of elements (including redundant zeros) in each row is a multiple of 16. This ensures a coalesced memory access. Figure 2 shows padding for slices 1 and 4 of the matrix shown in Figure 1. When a sparse matrix stored in SELLR-T format is multiplied by a vector (or a dense tall and skinny matrix), T threads $T = 2, 4, 8, \ldots$ work in parallel on the same row and compute partial sums.

To determine the format that works best in our application, we compared the performance of SELL-P and SELLR-T for

MOST WIEDZY Downloaded from mostwiedzy.pl

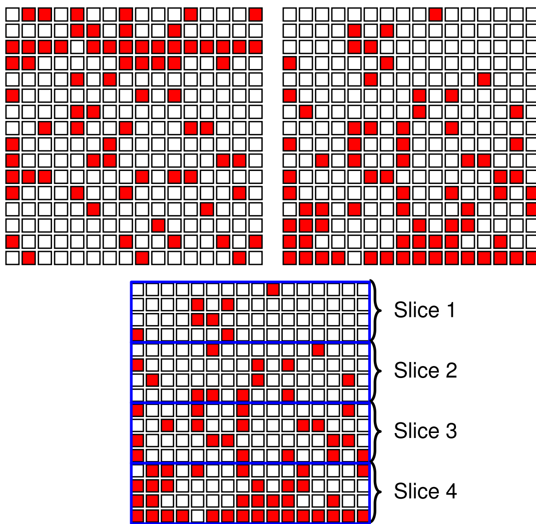

FIGURE 1. Matrix reordering and slicing .

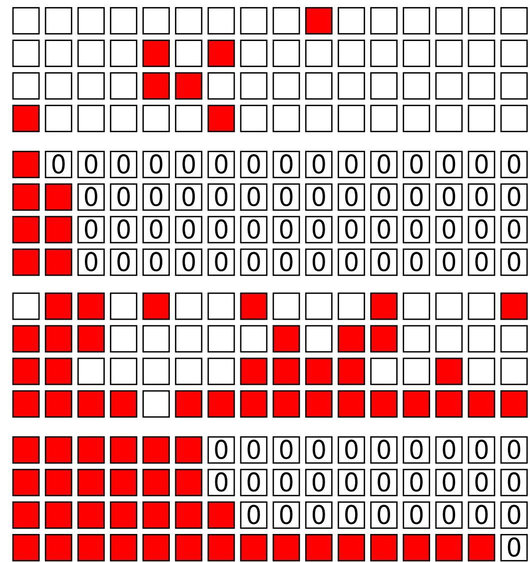

FIGURE 2. Zero padding in slices (slices 1 and 4) for a coalesced memory access.

our matrices on the same GPU as was used in the implementation of the GPU-accelerated LOBPCG solver for standard eigenvalue problems (i.e., when the mass matrix is the identity matrix) [11]. Tests reported in [30] on Tesla K40 accelerator (bandwidth 180 GB/s) showed that, for SpMV ($m = 1$), SELL-P achieved 82% of the upper bound performance (with a sparse matrix used [30]). Since Sliced ELLR-T format allowed us to achieve a slightly higher efficiency (86%) by our metrics, this format was used in the GPU-accelerated implementation of the sLOPBCG algorithm. On the NVIDIA P100 GPU, the SELLR-T format achieves 36.6 GFlop/s (m=1) or 111.3 (m=16) in the real double precision and 87.5 GFlop/s (m=1) or 243.6 GFlop/s (m=16) for the complex double precision SpMM kernel [9].

D. DUAL GPU IMPLEMENTATION

The SpMV and SpMM procedures were also developed for two P100 accelerators. As explained above, the main reason for using more than one accelerator was not the higher throughput, but to alleviate memory limitations. For a dual-GPU scenario, the stiffness and mass matrices were distributed between accelerators using a fast data-distribution technique that optimizes splitting of the finite-element method (FEM) matrices, as proposed in [8]. The technique uses graph partitioning on the level of tetrahedral FEM mesh and spreads the data between graphics accelerator. It also provides input to the FEM matrix-generation and assembly process needed to achieve load balancing and reduce communication between GPUs. In doing so, it also retains the blocked structure related to the order of the basis functions used in FEM, which is needed to use the hierarchical multilevel preconditioner. In particular, the 18 sparse stiffness and mass submatrices $\mathbb{K}_{ij}, \mathbb{M}_{ij}$, $i = 1, 3, j = 1, 3$ and PCG-V vectors were divided between two GPUs. The rest of sparse and dense matrices (the matrices associated with NSF and LOBPCG in Tab. 1) were stored in a host graphics accelerator. Consequently, one accelerator (the host) was loaded with more data than the other. This imbalance could be avoided, but then the performance would deteriorate significantly.

To provide reference results, the sLOBPCG algorithm was also implemented on a multicore CPU. For the CPU operation, whenever possible, we used parallelized procedures from the Intel MKL library (v. 2017, update 1). This library is optimized for performance on Intel's multicore processors, so we can regard the comparison of the runtimes involved in the tests as fair.

VI. NUMERICAL RESULTS

All numerical tests were executed on a server with an Intel Xeon (E5-2680 v3, 2.5 GHz, twelve cores) and 256 GB memory and one or two NVIDIA Tesla P100s (Pascal accelerator) with 3584 CUDA cores and 12 GB GPU RAM each. We considered a realistic electromagnetic problem—a dielectric resonator [17]. The structure is show in Figure 3

The resonator consists of a cylindrical cavity, loaded with a dielectric puck with a notch. The puck placed on a cylindrical support. The relative permeability μ_1 of the dielectric is equal to one, while the relative permittivity is a complex number given by

$$\epsilon_r = \epsilon' - j \tan \delta \tag{6}$$

where $\tan \delta$ is the loss tangent. The real part of the relative permittivity was assumed to be 37 and 2.1, for the puck and support, respectively. The loss tangent was varied in tests as described below.

Because of the presence of loss in the dielectric, the eigenvalues of the problem (1) are complex numbers. The real part is related to the resonant angular frequency ω_r, while the

MOST WIEDZY Downloaded from mostwiedzy.pl

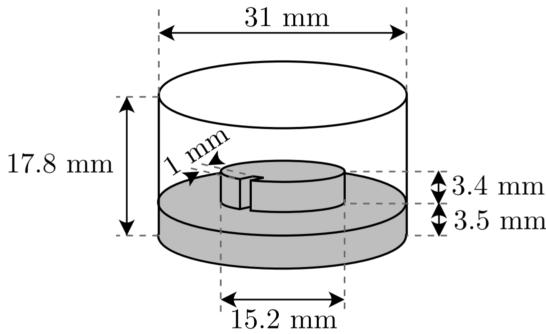

FIGURE 3. A resonant cavity loaded with lossy dielectrics .

TABLE 2. Test problem description.

Description	LOSSY1	LOSSY2
n	1 343 373	1 343 373
n_z	110 496 885	110 496 885
k	320 134	320 134
q	6	6
Loss tan δ (support)	10^{-3}	10^{-2}
Loss tan δ (resonator)	10^{-2}	10^{-1}
shift value (κ) in $\mathscr{P}$	8800	8800
shift value (γ) in sort	8800	8800
Q (the smallest ev)	103.4	10.34
$\epsilon_{sLOBPCG}$	10^{-4}	10^{-4}
ϵ_{PCG-V}	10^{-3}	10^{-10}
ϵ_{NSF}	10^{-6}	10^{-7}

imaginary part determines the quality factor Q.

$$\omega_r = c\mathrm{Re}\sqrt{\sigma} \tag{7}$$

$$Q = \frac{\omega_r}{(2\pi c\mathrm{Im}\sqrt{\sigma})} \tag{8}$$

We considered two cases: one in which the the loss tangent for the support and the dielectric pucks were assumed to be 0.001 and 0.01, respectively, and a second problem in which the losses were taken to be ten times higher. In the second case, the Q-factor for the lowest order mode is around 10, and this makes solving the eigenvalue problem challenging, even for some solvers utilized by industry [31]. In the case of SLOBPCG, the higher the loss, the more accurate the solution for the preconditioned residuals should be (step 11 in Algorithm 1). As a result, in order to find eigenvalues with the assumed tolerance ($\epsilon_{sLOBPCG}$) for the higher loss case, the requirements for accuracy of the sLOBPCG preconditioner ($\mathscr{P}$) are significantly higher (ϵ_{PCG-V}) for the first problem (10^{-10} vs. 10^{-3}). The FEM matrices used in the test were generated using InventSIM FEM [23] code. The tetrahedral mesh of the structure was generated using Netgen mesher [32]. Details of the two test problems analyzed in this paper are shown in Table 2. The complex eigenvalues σ, computed using sLOBPCG with the error tolerance set to $\epsilon = 10^{-4}$ for LOSSY1 and LOSSY2 setups, are presented in Tables 3 and 4. The eigenvalues found by our solver agree very well with the results calculated by commercial FEM software for solving microwave eigenvalue problems (the Eigenmode Solver of CST MICROWAVE STUDIO).

TABLE 3. Complex eigenvalues σ_j computed using sLOBPCG for error tolerance set to $\epsilon = 10^{-4}$ for LOSSY1.

Index j	σ_j	$\|\|\mathbb{K}\mathbf{X}_j - \mathbb{M}\mathbf{X}_j\sigma_j\|\|$
1	8893.627 + 85.996i	4.4e-07
2	12831.287 + 95.899i	2.5e-05
3	12954.640 + 93.342i	2.8e-05
4	15588.555 + 15.540i	8.7e-05
5	17541.405 + 98.468i	7.5e-05
6	17650.682 + 103.481i	1.9e-05

TABLE 4. Complex eigenvalues σ_j computed using sLOBPCG for error tolerance set to $\epsilon = 10^{-4}$ for LOSSY2.

Index j	σ_j	$\|\|\mathbb{K}\mathbf{X}_j - \mathbb{M}\mathbf{X}_j\sigma_j\|\|$
1	8812.187 + 852.111i	4.7e-05
2	12788.995 + 961.192i	1.1e-05
3	12918.337 + 935.799i	7.4e-06
4	15593.028 + 154.668i	2.5e-05
5	17428.701 + 966.968i	2.9e-05
6	17529.405 + 1016.704i	6.8e-05

TABLE 5. Memory (in GB) requirements and times (in seconds) taken by CPU-based sLOBPCG implementations for three different implementations of preconditioner $\mathscr{P}$.

$\mathscr{P}$	PCG-V	Pardiso-Seq	Pardiso-Sim
Memory [GB]	9.8	36.2	36.2
LOSSY1 [s]	814.9	805.1	516.7
LOSSY2 [s]	2633.2	964.9	664.5

The times taken to find six eigenpairs with a reference CPU-based sLOBPCG implementation (based on functions from Intel MKL in parallel mode) using the PCG-V solver for preconditioning and nullspace filtering, and with nonblocked SpMV products, were 815 and 2633 seconds for LOSSY1 and LOSSY2, respectively. To solve these eigenvalue problems, we needed 9.8 GB of CPU RAM. For higher levels of losses (LOSSY2), the tolerance needed in preconditioning residuals is high $\epsilon_{PCG-V} = 10^{-10}$, so it is in fact faster to use the direct solver (Intel Pardiso) for this operation, rather than the PCG-V solver. In this case, we take advantage of the symmetry of matrices and solve for all preconditioned residuals at once. The time taken by sLOBPCG on a CPU can be reduced from 2633 to 665 seconds. However, it should be borne in mind that the factorization-based Intel Pardiso requires about 3.7 times more memory than the PCG-V solver, and this grows rapidly as the matrix size increases, so this scenario is not a good reference for speed-up tests. In terms of memory usage, this variant is equivalent to the shift-and-invert approach, which is not intended for LOBPCG. The runtimes for various GPU-accelerated scenarios are given in Table 6 and Table 7. The tables also show the memory needed by each variant. As far as the GPU-memory usage is concerned, the reference is a GPU-accelerated implementation of Algorithm 1 with a nonblocked version of *SpMV* and the preconditioner stored as a separate matrix $\mathbb{A}$, rather than constructed on-the-fly from the blocks $\mathbb{K}_{ij}$, $\mathbb{M}_{ij}$, $i = 1, 3, j = 1, 3$. Such an implementation requires 11 GB of GPU RAM of 12 GB available on a

MOST WIEDZY Downloaded from mostwiedzy.pl

TABLE 6. Memory (in GB) requirements and time to solution (in seconds) for two reference implementation (one or two GPUs) of the GPU-accelerated sLOBPCG algorithm.

No. of GPUs	1	2	1	2
MatVec	SpMV	SpMV	SpMM	SpMM
Memory [GB]	11	8.5	> 12	> 12
LOSSY1 [s]	267.9	254.9	-	-
LOSSY2 [s]	738.8	675.5	-	-

TABLE 7. Memory (in GB) requirements and time to solution (in seconds) for GPU-memory-saving implementations of the sLOBPCG algorithm and two variant sparse matrix vector products (blocked SpMM, nonblocked SpMV).

No. of GPUs	1	2	1	2
MatVec	SpMV	SpMV	SpMM	SpMM
Memory [GB]	8.7	7.3	10.8	10.3
LOSSY1 [s]	294.9	281.3	178.0	179.1
LOSSY2 [s]	860.4	791.4	435.0	413.5

Pascal P100 (Table 6. For the dual-GPU scenario, the memory requirements drop to 8.5 GB (for dual GPU setups, we give the memory data for the host GPU, which is more loaded, as explained in Section IV: Implementation). For this reference GPU-accelerated implementation a nonblocked matrix–vector product, the time to solution is shortest for the dual GPU. Blocking of vectors for the sparse matrix–vector product (SpMM) was not possible, as there is not enough of GPU memory.

Storing 18 separate submatrices—$\mathbb{K}_{ij}, \mathbb{M}_{ij}$, $i = 1, 3$, $j = 1, 3$— in memory and then constructing the preconditioner from them reduces the requirements for nonblocked SpMV from 11 to 8.7 GB. Adding a second GPU allow a further memory reduction to 7.3 GB (on the host GPU). It can be noted that, while the memory requirements are reduced, the time to solution for this variant is longer. Comparing the runtime data in Table 6 and Table 7, it can be seen that, for the LOSSY2 case, the time taken by this GPU-memory-saving variant is 860.4 vs. 738.8 seconds, for a single GPU, and 791.4 vs. 675.5 seconds for the dual-GPU arrangement. This slowdown is because the matrix–vector products are carried out in the memory-saving version as a sequence of operations on individual blocks of the stiffness and mass matrices. However, the memory-saving implementation allows for blocking, which is very important for high performance. From Table 7, it is evident that a blocked sLOBPCG solver requires significantly more memory— 10.8 GB or 10.3 GB of GPU RAM for one or two graphics accelerators involved in computations. Nevertheless, comparing the runtimes, it is clear that the speed gains are large. The gains are particularly spectacular for the scenario with blocked vectors when many iterations of the preconditioner are needed (LOSSY2). The effect of blocking is clearly seen in Figure 4 which shows speedups for a single GPU setup.

With two GPUs, the solution takes 413.5 seconds, which is 6.4 times faster than the reference CPU implementation using the iterative preconditioner (PCG-V). The nonblocked

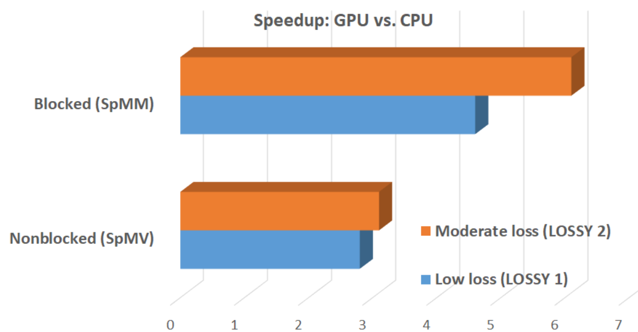

FIGURE 4. Speedup for blocked (SpMM) and nonblocked (SpMV) kernels for a single GPU setup.

TABLE 8. Comparison of GPU-based and CPU-based implementations of the sLOBPCG algorithm. Test problem: LOSSY1.

GPU (P100) ($\mathscr{P}$, MatVec)	CPU (Xeon) ($\mathscr{P}$, MatVec)	1 GPU vs. CPU	2 GPU vs. CPU
(PCG-V,SpMV)	(PCG-V,SpMV)	2.8	2.9
(PCG-V,SpMM)	(PCG-V,SpMV)	4.6	4.5

TABLE 9. Comparison of GPU-based and CPU-based implementations of the sLOBPCG algorithm. Test problem: LOSSY2.

GPU (P100) ($\mathscr{P}$, MatVec)	CPU (Xeon) ($\mathscr{P}$, MatVec)	1 GPU vs. CPU	2 GPU vs. CPU
(PCG-V,SpMV)	(PCG-V,SpMV)	3.1	3.3
(PCG-V,SpMM)	(PCG-V,SpMV)	6.1	6.4

version is 3.3 times faster than the CPU-implementation optimized for multicore platforms. For LOSSY1 problems, the speedups are lower—from 2.9 and 4.6 for the nonblocked and blocked vectors, respectively. The better speedup seen with LOSSY2 problems can be explained by the fact that LOSSY2 needs more iteration of the preconditioner and the preconditioner is the step of the sLOBPCG algorithm in which the gains from using a GPU are the highest. Concluding this comparison of the GPU-accelerated sLOBPCG algorithm with its parallel CPU-only counterpart, we can observe that GPU with an iterative preconditioner is faster even than the memory-demanding CPU implementation with a preconditioner based on the factorization of a sparse matrix and the direct solution of the system of equations. It is expected that even larger speedups can be obtained for the newest generation GPU accelerators. In our work, we used two NVIDIA Pascal P100 boards with 12 GB of RAM. The newest generation of NVIDIA's accelerators, Volta V100, supports 32 GB RAM, which would allow us to handle problem about three times larger, and also to obtain faster performance on nonblocked and blocked SpMV due to the much higher bandwidth and the increased number of cores.

VII. CONCLUSION

In this paper, fast single and dual GPU-based implementations of the stabilized locally optimal block preconditioned conjugate gradient (sLOBPCG) algorithm were presented.

MOST WIEDZY Downloaded from mostwiedzy.pl

Compared to the CPU reference implementation of the sLOBPCG algorithm, optimized for performance on a multicore processor using Intel MKL on the Intel Xeon (E5-2680 v3, 12 threads), the the GPU-accelerated code has been demonstrated to be 6.4 times faster in some computational scenarios. The approach proposed in the paper can be used to expedite finete-element modal analysis of microwave or photonic resonators in the presence of a lossy dielectric material.

REFERENCES

[1] M. Wang, H. Klie, M. Parashar, and H. Sudan, "Solving sparse linear systems on NVIDIA Tesla GPUs," in *Proc. Int. Conf. Comput. Sci.*, Berlin, Germany: Springer, 2009, pp. 864–873.
[2] S. Georgescu and H. Okuda, "Conjugate gradients on multiple GPUs," *Int. J. Numer. Methods Fluids*, vol. 64, nos. 10–12, pp. 1254–1273, 2010.
[3] A. Dziekonski, A. Lamecki, and M. Mrozowski, "GPU acceleration of multilevel solvers for analysis of microwave components with finite element method," *IEEE Microw. Wireless Compon. Lett.*, vol. 21, no. 1, pp. 1–3, Jan. 2011.
[4] A. Dziekonski, A. Lamecki, and M. Mrozowski, "Tuning a hybrid GPU-CPU V-cycle multilevel preconditioner for solving large real and complex systems of FEM equations," *IEEE Antennas Wireless Propag. Lett.*, vol. 10, pp. 619–622, 2011.
[5] R. Helfenstein and J. Koko, "Parallel preconditioned conjugate gradient algorithm on GPU," *J. Comput. Appl. Math.*, vol. 236, no. 15, pp. 3584–3590, 2012.
[6] R. Li and Y. Saad, "GPU-accelerated preconditioned iterative linear solvers," *J. Supercomput.*, vol. 63, no. 2, pp. 443–466, 2013.
[7] L. Z. Khodja, R. Couturier, A. Giersch, and J. M. Bahi, "Parallel sparse linear solver with GMRES method using minimization techniques of communications for GPU clusters," *J. Supercomput.*, vol. 69, no. 1, pp. 200–224, Jul. 2014.
[8] A. Dziekonski, P. Sypek, A. Lamecki, and M. Mrozowski, "Communication and load balancing optimization for finite element electromagnetic simulations using multi-GPU workstation," *IEEE Trans. Microw. Theory Techn.*, vol. 65, no. 8, pp. 2661–2671, Aug. 2017.
[9] A. Dziekonski and M. Mrozowski, "Block conjugate-gradient method with multilevel preconditioning and GPU acceleration for FEM problems in electromagnetics," *IEEE Antennas Wireless Propag. Lett.*, vol. 17, no. 6, pp. 1039–1042, Jun. 2018.
[10] J. L. Aurentz, V. Kalantzis, and Y. Saad, "Cucheb: A GPU implementation of the filtered lanczos procedure," *Comput. Phys. Commun.*, vol. 220, pp. 332–340, Nov. 2017.
[11] H. Anzt, S. Tomov, and J. Dongarra, "On the performance and energy efficiency of sparse linear algebra on GPUs," *Int. J. High Perform. Comput. Appl.*, vol. 31, no. 5, pp. 375–390, 2017.
[12] H. Anzt, S. Tomov, and J. Dongarra, "Accelerating the LOBPCG method on GPUs using a blocked sparse matrix vector product," in *Proc. Symp. High Perform. Comput.*, 2015, pp. 75–82.
[13] M. Kreutzer *et al.*, "Performance engineering and energy efficiency of building blocks for large, sparse eigenvalue computations on heterogeneous supercomputers," in *Software for Exascale Computing—SPPEXA*. Cham, Switzerland: Springer, 2016, pp. 317–338.
[14] M. Kreutzer *et al.*, "Chebyshev filter diagonalization on modern manycore processors and GPGPUs," in *Proc. Int. Conf. High Perform. Comput.*. Cham, Switzerland: Springer, 2018, pp. 329–349.
[15] W. Rodrigues, A. Pecchia, M. A. der Maur, and A. Di Carlo, "A comprehensive study of popular eigenvalue methods employed for quantum calculation of energy eigenstates in nanostructures using GPUs," *J. Comput. Electron.*, vol. 14, no. 2, pp. 593–603, 2015.
[16] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM J. Sci. Comput.*, vol. 23, no. 2, pp. 517–541, 2001.
[17] M. Rewienski, A. Dziekonski, A. Lamecki, and M. Mrozowski, "A stabilized complex LOBPCG eigensolver for the analysis of moderately lossy EM structures," *IEEE Microw. Wireless Compon. Lett.*, vol. 28, no. 1, pp. 7–9, Jan. 2018.
[18] A. Dziekonski, M. Rewienski, P. Sypek, A. Lamecki, and M. Mrozowski, "GPU-accelerated LOBPCG method with inexact null-space filtering for solving generalized eigenvalue problems in computational electromagnetics analysis with higher-order FEM," *Commun. Comput. Phys.*, vol. 22, no. 4, pp. 997–1014, 2017.
[19] P. Ingelstrom, "A new set of H (curl)-conforming hierarchical basis functions for tetrahedral meshes," *IEEE Trans. Microw. Theory Techn.*, vol. 54, no. 1, pp. 106–114, Jan. 2006.
[20] J. Rubio, J. Arroyo, and J. Zapata, "Analysis of passive microwave circuits by using a hybrid 2-D and 3-D finite-element mode-matching method," *IEEE Trans. Microw. Theory Techn.*, vol. 47, no. 9, pp. 1746–1749, Sep. 1999.
[21] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Philadelphia, PA, USA: SIAM, 2000.
[22] P. Arbenz and R. Geus, "Multilevel preconditioned iterative eigensolvers for Maxwell eigenvalue problems," *Appl. Numer. Math.*, vol. 54, no. 2, pp. 107–121, 2005.
[23] A. Lamecki, L. Balewski, and M. Mrozowski, "An efficient framework for fast computer aided design of microwave circuits based on the higher-order 3D finite-element method," *Radioengineering*, vol. 23, no. 4, pp. 970–978, 2014.
[24] S. Filippone, V. Cardellini, D. Barbieri, and A. Fanfarillo, "Sparse matrix-vector multiplication on GPGPUs," *ACM Trans. Math. Softw.*, vol. 43, no. 4, 2017, Art. no. 30.
[25] H. Anzt, S. Tomov, and J. Dongarra, "Implementing a sparse matrix vector product for the SELL-C/SELL-C-σ formats on NVIDIA GPUs," Innov. Comput. Lab., Dept. Elect. Eng. Comput. Sci., Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. UT-EECS-14-727, 2014.
[26] M. Clark, A. Strelchenko, A. Vaquero, M. Wagner, and E. Weinberg. (2017). "Pushing memory bandwidth limitations through efficient implementations of block-Krylov space solvers on GPUs." [Online]. Available: https://arxiv.org/abs/1710.09745
[27] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, "A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units," *SIAM J. Sci. Comput.*, vol. 36, no. 5, pp. C401–C423, 2014.
[28] A. Dziekonski, A. Lamecki, and M. Mrozowski, "A memory efficient and fast sparse matrix vector product on a GPU," *Prog. Electromagn. Res.*, vol. 16, pp. 49–63, 2011.
[29] F. Vázquez, G. Ortega, J. J. Fernández, and E. M. Garzón, "Improving the performance of the sparse matrix vector product with GPUs," in *Proc. 10th IEEE Int. Conf. Comput. Inf. Technol.*, Jun./Jul. 2010, pp. 1146–1151.
[30] A. Dziekonski and M. Mrozowski, "GPU acceleration of block Krylov methods for FEM problems in electromagnetics," in *Proc. IEEE MTT-S Int. Conf. Numer. Electromagn. Multiphys. Modeling Optim. RF, Microw., Terahertz Appl. (NEMO)*, May 2017, pp. 278–280.
[31] S. Papantonis and S. Lucyszyn, "Lossy spherical cavity resonators for stress-testing arbitrary 3D eigenmode solvers," *Prog. Electromagn. Res.*, vol. 151, pp. 151–167, 2015.
[32] J. Schöberl, "NETGEN an advancing front 2D/3D-mesh generator based on abstract rules," *Comput. Vis. Sci.*, vol. 1, no. 1, pp. 41–52, 1997.

ADAM DZIEKONSKI received the M.S.E.E. and Ph.D. degrees (Hons.) in microwave engineering from the Gdańsk University of Technology, Gdańsk, Poland, in 2009 and 2015, respectively. His current research interests include computational electromagnetics (mainly focused on parallelizing computations on graphics processing and central processing units). He was a recipient of the Domestic Grant for Young Scientists from the Foundation for Polish Science in 2012 and 2013. He was also a recipient of the Prime Minister's Award for his Ph.D. thesis in 2016.

MICHAL MROZOWSKI (F'08) received the M.Sc. and Ph.D. degrees (Hons.) from the Gdańsk University of Technology in 1983 and 1990, respectively. In 1986, he joined the Faculty of Electronics, Gdańsk University of Technology, where he is currently a Full Professor, the Head of the Department of Microwave and Antenna Engineering, and the Director of the Center of Excellence for Wireless Communication Engineering.

MOST WIEDZY Downloaded from mostwiedzy.pl