IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 8, 2018, accepted September 3, 2018, date of publication September 17, 2018, date of current version October 12, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2870737

Preconditioners With Low Memory Requirements for Higher-Order Finite-Element Method Applied to Solving Maxwell's Equations on Multicore CPUs and GPUs

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

Corresponding author: Grzegorz Fotyga (grzfotyg@pg.edu.pl)

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, cofinanced by the European Regional Development Fund, Smart Growth Operational Programme 2014–2020 through the Project EDISOn: Electromagnetic Design of flexIble SensOrs, and in part by the Faculty of Electronics, Telecommunications, and Informatics, Gdańsk University of Technology.

ABSTRACT This paper discusses two fast implementations of the conjugate gradient iterative method using a hierarchical multilevel preconditioner to solve the complex-valued, sparse systems obtained using the higher order finite-element method applied to the solution of the time-harmonic Maxwell equations. In the first implementation, denoted PCG-V, a classical V-cycle is applied and the system of equations on the lowest level is solved exactly. The second variant involves an approximate solution to the system of equations on the lowest level. To this end, auxiliary space preconditioning (ASP) is used instead of a direct solution. In this approach, denoted PCG-V-ASP, the time needed to solve the sparse system of equations is longer, but the memory requirements are smaller. To accelerate the computations, a graphics processing unit (GPU, Pascal P100) was used for both variants of the multilevel preconditioner. As a result, significant speedups were achieved over the reference parallel implementation using a multicore central processing unit (CPU, Intel Xeon E5-2680 v3, twelve cores). The results indicate that the auxiliary space preconditioning does in fact reduce the memory requirements, as compared with the reference PCG-V method, and at the same time performs each iteration faster. However, if symmetry is taken into account and the memory-efficient supernodal LDL^T factorization is employed, the savings are less spectacular than anticipated based on previously published results using LU factorization and the multifrontal technique. PCG-V also requires a fewer iterations, so it's time to solution is ultimately shorter. The difference is more pronounced if both preconditioners are run on a CPU. The use of a GPU as an accelerator for the computations considerably improves the performance of PCG-V-ASP over that of PCG-V.

INDEX TERMS Auxiliary space preconditioning, FEM, GPU, Maxwell's equations, multilevel preconditioning.

I. INTRODUCTION

The higher-order finite element method is one of the numerical tools most commonly used to solve complex electromagnetic problems. However, when the number of degrees of freedom is large, the solution of an FE system of equations becomes memory-demanding and time-consuming, especially when direct solvers are used. An efficient way to solve such problems with low memory requirements is to utilize Krylov-based iterative methods, such as the conjugate orthogonal conjugate gradient (COCG) [1]–[3] or the generalized minimal residual method (GMRES) [4], [5], with preconditioning to accelerate convergence. When applied to the solution of a sparse systems of equations derived from FEM discretization of the time-harmonic Maxwell's equations, many traditional preconditioners (e.g., those based on incomplete LU factorization) fail, and specialized preconditioners must be developed [6], [7].

Recently, a number of FEM preconditioners using higher-order basis functions have been proposed [6]–[11]. This idea draws on the assumption that the system matrix is partitioned into blocks of matrices associated with order of basis functions, providing a hierarchy of problems with associated

prolongation and restriction operations, which allow for a solution to be transferred from one level to the other. The preconditioner can thus be organized in a V or W cycle, known from multigrid methods, in such a way that a few iterations are applied at the highest level, and then the approximate solution from a higher level proceeds (via the intermediate levels) to the lowest, and then back to the highest. In the approaches proposed in [6] (PCG-V) and [7] (pMUS), the problem on the lowest level is solved using a factorization-based direct solver for sparse linear systems of equations. Unlike these approaches, the auxiliary space preconditioning (ASP) method, introduced in [12] and developed further in [11], [13], and [14], assumes that the problem on the lowest level is approximately solved in auxiliary spaces (a space of piecewise linear scalar functions and a space of nodal vector functions) defined on the same tetrahedral mesh. This strategy guarantees that RAM demands are significantly lower than in factorization-based direct solutions on the lowest level of the multilevel preconditioner; it is thus especially useful in solving large problems. However, it should be borne in mind that the remarkable benefits of the ASP approach reported in [11], [13], and [14], were achieved using the UMFPACK [15], [16] direct solver for sparse systems of equations which, in our tests, has proved to be a highly memory-demanding library. UMFPACK is based on the multifrontal factorization technique. Numerical experiments show that other methods for factorizing a sparse matrix, such as the supernodal-based [17] technique implemented in PARDISO solvers, require much less memory than UMFPACK, so it is not clear to what extent the ASP preconditioner reduces the memory requirements when used with a memory-efficient direct solver for sparse systems of linear equations.

In this paper, we present and compare the performance of the conjugate gradient method with two preconditioners: a V-cycle multilevel preconditioner (PCG-V) and a V-cycle multilevel preconditioner with auxiliary space preconditioning (PCG-V-ASP). In order to solve the relatively small sparse systems of equations that occur in both preconditioners, we employ the supernodal sparse Pardiso solver available in the Intel MKL library and use LDL^T factorization, rather than the multifrontal-based UMFPACK solver with LU factorization used previously in [11], [13], and [14]. This choice is intended to provide a better picture of where the ASP-based preconditioner stands with respect to the regular PCG-V preconditioner. The preconditioners considered can be implemented in such a way that various computations are performed concurrently. Initially, computations in both solvers are performed on a central processing unit (CPU), with the Intel MKL functions executed in parallel mode. These tests show that, because the ASP preconditioner needs more iterations to converge, it requires significantly more time to solve a sparse system of equations than when the PCG-V solver is applied. On the other hand, ASP involves fewer operations that do not scale well with an increasing number of cores. To determine whether increased parallelism can offset the larger number of iterations, we have also developed code for both solvers that makes use of a GPU accelerator with as many as 3584 cores, and compares the performance of preconditioners for this GPU-accelerated scenario with the CPU-only multithreaded code optimized for multicore processors.

The paper is organized as follows: Section II briefly describes the finite element formulations used to compute the scattering parameters of the structure being examined. Two preconditioning techniques are described in Section III. Implementations of the proposed approaches are described in Section IV. Finally, our results are presented and discussed in Section V.

II. FINITE ELEMENT METHOD

Let us consider a lossy dielectric-loaded structure Ω enclosed by a boundary S. The distribution of the electric field in Ω is determined 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, $k_0 = \omega\sqrt{\mu_0 \epsilon_0}$ is the wavenumber, and μ_r and ϵ_r are the relative permittivity and permeability, respectively. Assuming that the structure is excited through m ports (with a single mode excitation in each of the ports), the weak formulation of (1) reads [18]:

$$\int_\Omega \left(\nabla \times \vec{w} \cdot \frac{1}{\mu_r} \nabla \times \vec{E} - k_0^2 \vec{w} \cdot \epsilon_r \vec{E} \right) d\Omega - j\omega\epsilon_0 \sum_{i=1}^{m} \int_{P_i} \vec{w} \cdot (\vec{n}_i \times \vec{H}_{ti}) dP_i = 0, \tag{2}$$

where ϵ_0 is the absolute permittivity, P_i is the surface of the i-th port, $\vec{n}_i$ is a unit vector normal to P_i, $\vec{w}$ is a vector testing function, ω is the angular frequency, j is the imaginary unit, and $\vec{H}_{ti}$ is a distribution of the tangential magnetic field at P_i.

A 3D vector finite element method formulation with hierarchical vector basis functions up to the third order [9] was chosen to solve (2). Following the standard FEM procedure [18], we obtain:

$$(\mathbb{S} - k_0^2 \mathbb{T})\mathbf{E} = j\omega \bar{\mathbf{B}} \bar{\mathbf{I}}_m, \tag{3}$$

where $\mathbb{S}, \mathbb{T} \in \mathbb{C}^{n \times n}$ are sparse stiffness and mass matrices, respectively, n is the number of degrees of freedom (DoF), $\bar{\mathbf{I}}_m$ is a diagonal matrix with m amplitudes of waveguide modes, and $\bar{\mathbf{B}}$ consists of n-dimensional vectors: $\bar{\mathbf{B}} = \left[\bar{\mathbf{b}}_1, \bar{\mathbf{b}}_2 \ \ldots \ \bar{\mathbf{b}}_m\right]$. The elements of vector $\bar{\mathbf{b}}_i$ (associated with the i-th port) are computed as follows:

$$\bar{\mathbf{b}}_i = \begin{bmatrix} \int_{P_i} \vec{T}_1 \cdot (\vec{n}_i \times \vec{H}_{ti}) dP_i \\ \ldots \\ \int_{P_i} \vec{T}_n \cdot (\vec{n}_i \times \vec{H}_{ti}) dP_i \end{bmatrix}. \tag{4}$$

The next step involves the normalization of $\bar{\mathbf{B}}$ and $\bar{\mathbf{I}}_m$ with respect to the characteristic impedance of each of m ports:

$$\mathbf{I}_m = \bar{\mathbf{I}}_m \mathbf{Z_P}^{\frac{1}{2}}, \quad \mathbf{B} = c\bar{\mathbf{B}}/\mathbf{Z_P}^{-\frac{1}{2}}, \tag{5}$$

$\mathbf{Z_P} = diag(Z_{P1}, Z_{P2} \ldots Z_{Pm})$ is a diagonal matrix containing the characteristic impedances. Substituting (5) into (3) yields the final form of the system of equations:

$$(\mathbb{S} - k_0^2\mathbb{T})\mathbf{E} = jk_0\mathbf{B}\mathbf{I}_m. \tag{6}$$

where the system matrix $\mathbb{A} = \mathbb{S} - k_0^2\mathbb{T}$ is sparse, symmetric, and indefinite.

Next, the system of equations (6) is solved in order to compute the desired parameters of the structure. For large problems, it is worthwhile (in terms of memory usage and computation time) to use iterative, Krylov-subspace based methods, such as COCG [1] or GMRES [4]. For nonpositive definite systems (6), the conjugate gradient method (CG) cannot be applied. However, the utilization of multilevel or multigrid preconditioners with the CG method is common in electromagnetics [6].

Finally, the scattering matrix ($\mathbf{S}$) of the structure is obtained as follows:

$$\mathbf{S}(k_0) = 2(\mathbf{I} + \mathbf{Y}(k_0))^{-1} - \mathbf{I}, \tag{7}$$

where $\mathbf{I}$ is the identity matrix, $\mathbf{Y}(k_0) = \mathbf{Z}(k_0)^{-1}$, and $\mathbf{Z}(k_0) = \mathbf{B}^T\mathbf{E}$ is an input-to-output transfer function.

III. SOLUTION OF A SPARSE SYSTEM OF EQUATIONS

The system matrix that emerges from FEM with higher-order basis functions is large and sparse. The direct solution of the system of equations involves factorization of the sparse matrix. Unfortunately, the factors are much denser than the matrix that is being factorized. As the number of degrees of freedom grows, the number of nonzero elements in the factors increases rapidly, and a direct solution of the system of equations becomes impossible because of the limitations of CPU memory.

To roughly determine the practical limit of direct solvers in computational electromagnetics when FEM higher-order basis functions are used to solve the time-harmonic Maxwell's equations, we considered two software libraries designed for the solution of sparse linear systems: UMFPACK and Intel MKL Pardiso. UMFPACK [15] is a popular numerical library, also used in Matlab, for LU factorization of a sparse matrix and solving sparse systems of equations. The code uses column preordering [19] to reduce fill-in, followed by symbolic factorization and a right-looking unsymmetric-pattern multifrontal numerical factorization. Although the matrix pattern analysis and permutations are intended to reduce memory requirements, the actual memory usage is rather high. Our test, carried out with higher-order FEM matrices resulting from simulations of a lossy filter, have shown that the factorization of a complex-valued sparse matrix with 1.2 million unknowns with UMFPACK would need over 2 TB of RAM. This figure is clearly too large for any practical use. It is also worth noting that UMFPACK does not take the symmetry of the matrix into account. In FEM, the resulting matrix is indefinite but symmetric, so the more efficient LDL^T factorization should be applied. However, even if symmetry is not considered, better results in terms of memory consumption can be achieved if sparse LU factorization is carried out using a supernodal approach [17], as adopted by the Intel MKL Pardiso solver for directly solving sparse systems of equations. For the same problems, the Intel MKL Pardiso solver needs much less memory than UMFPACK. Table 1 shows the memory requirements for storing and factorizing complex-valued matrices with different number of rows when the number of tetrahedra grows from 66,735 to 277,143. Complex-valued FEM matrices were generated using the FEM code described in [20], assuming the basis functions up to the third-order. For a problem with 1.2 million degrees of freedom. Pardiso needs 32 GB, two orders of magnitude less than UMFPACK. The difference in memory requirements is remarkable, and the use of UMFPACK should be restricted to very small matrices. It can also be seen from the table that the memory requirements of Intel MKL Pardiso grow very rapidly. LU factorization of a system with 5.1 million unknowns was not possible on a workstation equipped with 256 GB RAM. Taking advantage of the symmetry of FEM matrices, and using PARDISO's LDL^T factorization capabilities (not available in UMFPACK), brings about considerable savings, as shown in Table 2, where the memory requirements are given for matrices stored in symmetric form and for LDL^T factorization (rather than LU).

TABLE 1. Memory (in GB) required to store the sparse matrix $\mathbb{A}$ and to factorize a complex-valued matrix $\mathbb{A}_{11}$ with Intel MKL Pardiso. LU factorization: sparse matrices are stored as general.

Tetrahedra	rows (mln)	A	Factorization	sum
66 735	1.2	2.0	32.0	34.0
124 303	2.2	3.7	54.3	57.9
277 143	5.1	8.7	>256	>256

TABLE 2. Memory (in GB) required to store the sparse matrix $\mathbb{A}$ and to factorize a complex-valued matrix $\mathbb{A}_{11}$ with Intel MKL Pardiso. LDL^T factorization: sparse matrices are stored in upper triangular form.

Tetrahedra	rows (mln)	A	Factorization	sum
66 735	1.2	1.0	18.8	19.8
124 303	2.2	1.9	31.7	33.6
277 143	5.1	4.4	127.1	131.5

A. PRECONDITIONED ITERATIVE SOLVERS

When memory requirements grow with problem size, the direct solution of larger problems becomes impossible and iterative Krylov-space methods must be applied. Iterative solvers are memory-efficient, but their convergence may be poor. Preconditioning is needed to improve the convergence rate. Preconditioning transforms the original linear system into one which has the same solution, but is likely to be easier to solve with an iterative solver [21]. Ideally the

application of a preconditioner in each iteration should be equivalent to the multiplication of a system matrix by its (approximate) inverse. The preconditioner itself can be a matrix or a sequence of operations. A preconditioner requires extra memory, which must be taken into account when selecting a preconditioning technique. Also, modern workstations have multicore CPUs and are often equipped with accelerators, such as graphics processing units, which contain thousands of cores. If these computational resources can be used in parallel, the time to solution can be significantly reduced.

This section introduces two preconditioners with low memory requirements and good parallelization properties, specifically designed for the finite element method (FEM) with higher-order vector basis functions.

1) MULTILEVEL PRECONDITIONER

A hierarchical multilevel preconditioner (Hie-ML) is a set of operations (Listing 1), which can have different schemes (V-cycle, W-cycle) and, depending on the order of basis functions, may have several levels. In our case, basis functions up to the third order have been used (QTCuN) [9], so we can employ three levels. 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. In our case, the division is as follows:

$$\begin{bmatrix} \mathbb{A}_{11} & \mathbb{A}_{12} & \mathbb{A}_{13} \\ \mathbb{A}_{21} & \mathbb{A}_{22} & \mathbb{A}_{23} \\ \mathbb{A}_{31} & \mathbb{A}_{32} & \mathbb{A}_{33} \end{bmatrix} = \begin{bmatrix} \mathbb{S}_{11} & \mathbb{S}_{12} & \mathbb{S}_{13} \\ \mathbb{S}_{21} & \mathbb{S}_{22} & \mathbb{S}_{23} \\ \mathbb{S}_{31} & \mathbb{S}_{32} & \mathbb{S}_{33} \end{bmatrix} - k_0^2 \begin{bmatrix} \mathbb{T}_{11} & \mathbb{T}_{12} & \mathbb{T}_{13} \\ \mathbb{T}_{21} & \mathbb{T}_{22} & \mathbb{T}_{23} \\ \mathbb{T}_{31} & \mathbb{T}_{32} & \mathbb{T}_{33}, \end{bmatrix} \quad (8)$$

where $\mathbb{A}_{11}$ corresponds to the Whitney basis functions.

```
1  z = Hie-ML(r, i)
2  z = 0
3  if i == 1 then
4     z = A_11^{-1} r  // solve - the lowest level
5  else
6     smoothing(z, r)  //the highest level
7     r^{i-1} = r - A_{i-1,i} z
8     z^{i-1} = Hie-ML(r, i-1)
9     r^i = r - A_{i,i-1} z^{i-1}
10    smoothing(z, r)  //the highest level
```

Listing 1. Hierarchical multilevel preconditioner (*Hie-ML*).

Listing 1 shows that the Hie-ML preconditioner has smoothing operations (lines 6, 10), a sparse matrix vector products between levels (lines 7, 9), and a solution of the sparse system of linear equations on the lowest level (line 4). To perform smoothing operations, relaxation iterative methods can be used. Zhu and Cangellaris [6] proposed the Gauss–Seidel method for smoothing operators, in order to guarantee satisfactory convergence. However, the Gauss–Seidel method cannot be efficiently parallelized, meaning that high performance cannot be achieved. Thus, a weighted Jacobi method (*wJacobi*) was proposed [10], based on the sparse matrix vector product, and thus capable of easy parallelization. Finally, the system of equations on the lowest level of the preconditioner is solved by means of the direct solver [22]. The memory requirements of the preconditioner are associated with the memory needed for factorizing and storing factors of a sparse matrix $\mathbb{A}_{11}$ (associated with the lowest-order FEM basis functions). This matrix is much smaller than the whole system matrix $\mathbb{A}$, so the memory requirements are lower.

2) AUXILIARY SPACE PRECONDITIONING

The memory requirements of a hierarchical preconditioner can be reduced even further if the system of equations on the lowest level of a hierarchical preconditioner (Listing 1, line 4) can be solved by means of the auxiliary space preconditioner (ASP) technique [13]. This takes advantage of the three spaces of basis functions:

- $\mathcal{V}$: the space spanned by the first order vector basis functions (Whitney functions [9]), tangentially continuous, associated with the edges of the mesh,
- $\mathcal{N}$: an auxiliary space of linear scalar basis functions. $\nabla\mathcal{N}$ is a subspace of $\mathcal{V}$, assuming the same mesh.
- $\mathcal{N}^3$: an auxiliary space of nodal vector functions, which are tangentially and normally continuous.

The functions from $\mathcal{N}$ space can be projected onto $\mathcal{V}$ space by means of the *node-to-edge* sparse mapping matrix $\mathbb{G}$, whereas the transposed matrix $\mathbb{G}^T$ is used for the reverse mapping from $\mathcal{V}$ to $\mathcal{N}$. In the same way, functions from $\mathcal{N}^3$ space can be projected onto $\mathcal{V}$ using a sparse Π matrix, composed of the three blocks associated with the Cartesian coordinates:

$$\Pi = \begin{bmatrix} \Pi_x & \Pi_y & \Pi_z \end{bmatrix}, \quad (9)$$

while Π^T is used for the reverse mapping. The details of the implementation of Π and $\mathbb{G}$ can be found in [23] and [24].

The system matrix $\mathbb{A}$ associated with the functions from $\mathcal{V}$ space is represented in the two auxiliary spaces $\mathcal{N}$ and $\mathcal{N}^3$ by means of the square matrices:

$$\mathbb{A}_{11}^n = \mathbb{G}^T \mathbb{A}_{11} \mathbb{G} \quad (10)$$

and:

$$\mathbb{A}_{11}^\zeta = \Pi_\zeta^T \mathbb{A}_{11} \Pi_\zeta \quad (11)$$

where $\zeta = \{x, y, z\}$.

Similarly to in [11], we have utilized the shifted Laplacian technique, in which it is assumed that the matrices $\mathbb{A}_{11}^x$, $\mathbb{A}_{11}^y$, and $\mathbb{A}_{11}^z$ are constructed in such a way that the term k_0^2, used in the volume integral of (2), is replaced by $(1 - j\gamma)k_0^2$, where $\gamma = 0.8$. In effect, the eigenvalues of the resulting preconditioned matrix are clustered in the right-hand half of the complex plane, which significantly improves the performance of ASP (see [11] for details).

```
1  z = ASP(r, i)
2  Δz = smoothing(z, r); z = Δz
3  r = r − 𝔸11 Δz // Residual update
4  r^c = 𝔾^T r // Transfer to subspace
5  Δz^c = (𝔸11^n)^{-1} r^c // Solve
6  Δz = 𝔾 Δz^c; z = z + Δz // Transfer back
7  for ζ = x, y, z
8      r^c = Π_ζ^T r // Transfer to subspaces
9      Δz^c = (𝔸11^ζ)^{-1} r^c // Solve
10     Δz = Δz + Π_ζ Δz^c // Transfer back
11     z = z + Δz
12 r = r − 𝔸11 Δz // Residual update
13 Δz = smoothing(z, r); z = z + Δz
```

Listing 2. Auxiliary space Preconditioner (*ASP*), V-cycle.

The solution of the system of equations on the lowest level of a hierarchical preconditioner can be approximated by the following formula:

$$\mathbb{A}_{11}^{-1}\mathbf{r} \cong \text{smoothing}(\mathbf{z}, \mathbf{r}) + \mathbb{G}(\mathbb{A}_{11}^{n})^{-1}\mathbb{G}^{T}\mathbf{r} + \sum_{\zeta=x,y,z} \Pi_{\zeta}(\mathbb{A}_{11}^{\zeta})^{-1}\Pi_{\zeta}^{T}\mathbf{r}. \quad (12)$$

where the smoothing procedure is performed by using a weighted Jacobi method (similar to in the case of Hie-ML). The formula (12) can be expressed in the form of a V-cycle scheme, as shown in Listing 2.

In ASP, the majority of memory usage is related to factorization, and thus to sparse matrices (10) and (12)—see Listing 2, lines 5 and 9. These matrices are much smaller than matrix $\mathbb{A}_{11}$, which appears in the standard hierarchical preconditioner (Listing 1, line 4).

TABLE 3. Description of the CPU-based and GPU-based implementations.

Operation	CPU-based	GPU-based
SpMV	CPU (Intel MKL)	GPU (SELLR-T [25])
BLAS1 (axpy,cpy)	CPU (Intel MKL)	GPU (CUDA)
direct solvers	CPU (Intel PARDISO)	CPU (Intel PARDISO)

IV. IMPLEMENTATION

This paper compares two implementations: The reference (CPU-based) implementation uses the Intel Math Kernel Library (Table 3), in which computations are performed in parallel using all CPU cores. The library is tuned for the fastest operation of Intel multicore processors, so we consider our CPU-only version of the preconditioned PCG to be a good yardstick for comparison with GPU-accelerated solvers. In the second, GPU-based, implementation most of the computations (a sparse matrix vector product, BLAS1 operations) are massively parallelized on a graphics accelerator, and only direct solutions on the lowest level of the V-cycle preconditioners are performed on a CPU, using the Intel MKL PARDISO solver. More precisely, in the GPU-based implementation, a CPU is used to carry out computations in line 4 (Listing 1) of the standard hierarchical multilevel preconditioner (*Hie-ML*, or on lines 5 and 9 (Listing 2) of the ASP preconditioner. This requires communication between the GPU and CPU. However, the overhead due to the data transfer is not significant, since the size of the matrices $\mathbb{A}_{11}$ (V) and $\mathbb{A}_{11}^{n}$ and $\mathbb{A}_{11}^{\zeta}$ (V-ASP) is relatively small compared to $\mathbb{A}$. In the case of PCG-V, there are more transfers between CPU and GPU per iteration, as steps 2–4, 6, 8, and 10–13 in (Listing 2) are executed on a GPU; however, since ASP operates on very small matrices, this does not affect the performance much.

We used Intel MKL (2017, update 1) for the CPU operations. To implement the GPU operations, cuSPARSE (v8.0) was applied whenever possible—except for the sparse-matrix vector product (SpMV), for which we developed our own computational kernels. In these kernels, a sparse matrix is stored in the Sliced ELLR-T sparse matrix format [25]. In this format, the matrix is divided into slices, nonzero entries are permuted in each slice, and rows with fewer nonzero elements are zero-padded to ensure coalesced access to data. This format yields much better performance than the standard CRS format employed in the cuSPARSE library. More precisely, for the hardware used in the test, we obtained almost 90 GFlops for complex-valued sparse matrix, compared to the 65 GFlops achievable with cuSPARSE [26]. To save GPU memory, the system matrix $\mathbb{A}$ is divided into nine blocks; the SpMV operations required in various stages of the PCG algorithm and preconditioner are performed on a GPU and involve one or more such blocks (see [10] for details).

The upper performance bound (roofline model) for the SpMV kernel is given by:

$$P_{max} = \frac{Flops}{t_B} BW, \quad (13)$$

where *Flops* is the number of SpMV floating point operations, t_B is lower bound on data transfer (in bytes), and BW is the maximal data transfer from/to main memory.

V. NUMERICAL RESULTS

All numerical tests were executed on an Intel Xeon (E5-2680 v3, 2.5 GHz, twelve cores) with 256 GB memory and an NVIDIA Tesla P100 (Pascal accelerator) with 3584 CUDA cores and 12 GB GPU RAM. To investigate the behavior of the PCG-V and PCG-V-ASP methods, we considered a realistic lossy electromagnetic problem, in the form of a four-pole dielectric-loaded cavity filter [27], shown in Fig. 1. In each resonator cavity, large cylindrical dielectric pucks with dielectric relative permittivity $\epsilon_r = 30 + j3 \times 10^{-6}$ are located on dielectric supports (with $\epsilon_r = 9$). All the other filter parameters are provided in [27]. Complex-valued FEM matrices were generated using the higher-order FEM code described in [20]. The number of weighed Jacobi iterations in the presmoothing and postsmoothing phases of the multilevel preconditioner (both in PCG-V and PCG-V-ASP) are 2 and 7, in the third and second levels, respectively.

As we observed in the introduction, the memory requirements determine the practical use of a given solver on a particular workstation. In previous papers with auxiliary space preconditioning [11], [13], [14], UMFPACK code was used as a direct solver for sparse systems of equations. Here, Intel MKL Pardiso is used instead. We selected for our tests

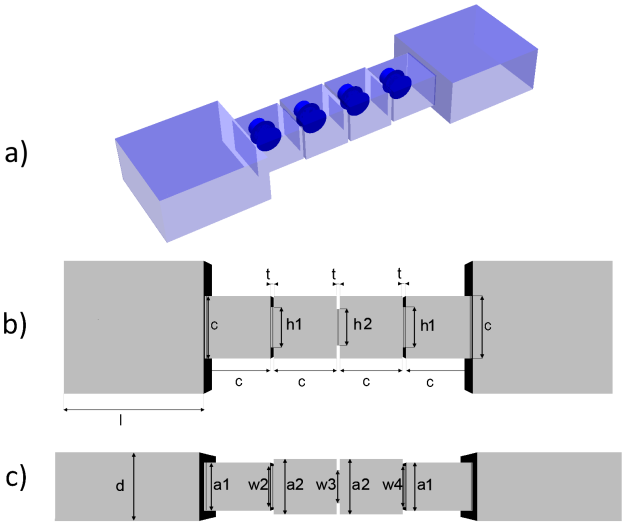

FIGURE 1. a) Four-pole dielectric-loaded filter. b) and c) top and side view of the structure with the dimensions (in mm): $a_1 = 6.91$, $a_2 = 7.93$, $c = 9.0$, $w_2 = w_4 = 5.93$, $w_3 = 4.85$, $h_1 = 5.86$, $h_2 = 5.25$, $t = 0.5$, $d = 9.52$, $l = 20.0$, the radius, height and the permittivity of the pucks: 2.55, 2.30, 30 and 1.75, 2.31, 9, respectively.

TABLE 4. Properties of the largest test problem analyzed in the paper. $\mathbb{A}_{11}^{n}$ and $\mathbb{A}_{11}^{\zeta}$ were defined in Eqs. (10)-(11), respectively.

Matrix	Preconditioner	rows	cols	nnz
$\mathbb{A}$	-	5 100 339	5 100 339	432 993 471
$\mathbb{A}_{11}$	V	320 184	320 184	5 185 808
$\mathbb{A}_{11}^{n}$,$\mathbb{A}_{11}^{\zeta}$	V-ASP	45 935	45 935	663 175

the largest problem, with 5.1 million degrees of freedom; Table 4 shows the characteristics of the most important matrices involved in the preconditioners. It can be seen that the matrices to be factorized are much smaller than the matrix $\mathbb{A}$. Also, the matrices used in the ASP preconditioner are about ten times smaller than the matrix $\mathbb{A}_{11}$ used on the lowest level of the standard multilevel preconditioner (see line 4 in Listing 1). In the GPU-based version of the preconditioner, all steps except for the direct solution of the sparse system on the lowest level are carried out on the GPU; the size of the matrix on the lowest level is also important from the point of view of communications between the CPU and GPU. If a smaller matrix in the ASP preconditioner is processed on the CPU, less data traffic occurs upon each iteration. This implies that the PCG-V-AS solver should benefit more than the PCG-V solver from the increased number-crunching power provided by the GPU.

TABLE 5. Memory (in GB) required to store the sparse matrix $\mathbb{A}$ using LU factorization of the matrix $\mathbb{A}_{11}$ with Intel MKL Pardiso and vectors in the iterative solver PCG-V.

rows (mln)	$\mathbb{A}$	$\mathbb{A}_{11}$ Fact.	Vectors	sum
1.2	2.0	0.6	0.3	2.9
2.2	3.7	1.0	0.5	5.2
5.1	8.7	4.5	1.2	14.4

Table 5 shows the amount of memory required to store the sparse matrix $\mathbb{A}$, and to factorize and store the factors of the matrix $\mathbb{A}_{11}$ with the Intel MKL Pardiso used in the iterative PCG-V solver. The storage needed for vectors is also given for completeness. It is worth noting that, even if the $\mathbb{A}_{11}$ matrix has about 17 times fewer rows, its factorization is memory-consuming and utilizes as much as 31% of the total memory requirements of the PCG-V solver.

TABLE 6. Memory (in GB) required to store the sparse matrix $\mathbb{A}$ using LU factorization of the matrices $\mathbb{A}_{11}^{n}$ and $\mathbb{A}_{11}^{\zeta}$ with Intel MKL Pardiso and vectors in the iterative solver PCG-V-ASP.

rows (mln)	$\mathbb{A}$	ASP	Vectors	sum
1.2	2.0	0.2	0.3	2.5
2.2	3.7	0.3	0.5	4.6
5.1	8.7	1.5	1.2	11.5

TABLE 7. Memory (in GB) required to store the sparse matrix $\mathbb{A}$ using LDL^T factorization of the matrix $\mathbb{A}_{11}$ with Intel MKL Pardiso and vectors in the iterative solver PCG-V.

rows (mln)	$\mathbb{A}$	$\mathbb{A}_{11}$ Fact.	Vectors	sum
1.2	2.0	0.4	0.3	2.7
2.2	3.7	0.6	0.5	4.9
5.1	8.7	2.7	1.2	12.6

TABLE 8. Memory (in GB) required to store the sparse matrix $\mathbb{A}$ using LDL^T factorization of the matrices $\mathbb{A}_{11}^{n}$ and $\mathbb{A}_{11}^{\zeta}$ with Intel MKL Pardiso and vectors in the iterative solver PCG-V-ASP.

rows (mln)	$\mathbb{A}$	ASP	Vectors	sum
1.2	2.0	0.2	0.3	2.4
2.2	3.7	0.2	0.5	4.5
5.1	8.7	1.1	1.2	11.0

The PCG-V-ASP solver (Table 6) with LU factorization carried out using Intel Pardiso requires significantly less memory than PCG-V to solve the system on the lowest level. Comparing the data in the third column of Table 5 and Table 6, it can be seen that the factorization of matrices on the lowest level, and then storing the factors, requires three times as much memory in the standard PCG-V as in PCG-V-ASP. Overall, the relative CPU-memory savings are less spectacular. For the system with 5.1 million unknowns, the memory savings in the PCG solver with the auxiliary space preconditioner amount to 26% (a reduction from 14.4 GB to 11.5 GB). If the symmetry of the FEM matrices is taken advantage of and LDL^T factorization is applied, the memory requirements of the iterative solvers are obviously lower, but the difference here between PCG-V and PCG-V-ASP is not very significant. Tables 7 and 8 show that PCG-V and PCG-V-ASP require in total 12.6 GB and 11.0 GB, respectively. This is a good result, but the savings are not as dramatic as reported in [11], [13], and [14], where a memory-hungry LU factorization using UMFPACK was considered. It is worth noting that the memory consumption for factorizing matrices used in the multilevel preconditioner are significantly lower in Intel MKL Pardiso than in UMFPACK. In particular, Pardiso requires about 20 and 10 times less memory to factorize $\mathbb{A}_{11}$ in PCG-V and $\mathbb{A}_{11}^{n}$ in PCG-V-ASP, respectively.

It is evident that PCG-V-ASP requires less memory than the standard PCG-V; however, since the difference is not very significant, it is also important to compare both preconditioners in terms of time needed to solve the system.

To this end, we tested two parallel implementations of the preconditioned conjugate solver with LDL^T factorization, using the largest problem involving a complex-valued matrix with 5.1 million rows. The first implementation uses only a multicore CPU. The second implementation takes advantage of a GPU accelerator. The average times taken by the singe iteration of the PCG-V and PCG-V-ASP solvers for a CPU-only implementation are presented in the second to sixth rows of Table 9. While the operations in the main CG iteration and the smoothing in the preconditioner take a similar amount of time, there is a significant time difference in favor of PCG-V-ASP for the solution on the lowest level of the preconditioner, as seen from the fifth row in the table. The time taken to solve the system on the lowest level is not dominant. As a result, the average time taken by a single iteration of PCG-V-ASP is only about about 7% shorter for PCG-V-ASP. Unfortunately, the PCG-V-ASP solver requires over three times as many iterations as PCG-V. Based on the results of the tests, it can be concluded that, despite its lower memory requirements, the savings with LDL^T factorization are not significant from the performance point of view. It is thus preferable to use the PCG-V solver.

TABLE 9. Comparison of the PCG-V and PCG-V-ASP solvers for a CPU-based implementation. LDL^T factorization of sparse matrices on the lowest level ($\mathbb{A}_{11}$ in PCG-V; $\mathbb{A}_{11}^n$ and $\mathbb{A}_{11}^\zeta$ in PCG-V-ASP).

Operation	PCG-V [s]	PCG-V-ASP [s]	Speedup
CG (1 it.)	0.58	0.53	1.1
V (1 it.)	1.89	1.78	1.1
V-smoothing (1 it.)	1.49	1.57	1.0
V-lowest level (1 it.)	0.40	0.21	1.9
Solver (1 it.)	2.48	2.30	1.1
No. of iters. ($\epsilon < 1e^{-6}$)	27	101	0.3
Solver (all)	66.88	232.53	0.3

TABLE 10. Comparison of the PCG-V and PCG-V-ASP solvers for a GPU-based implementation. LDL^T factorization of sparse matrices ($\mathbb{A}_{11}$ in PCG-V; $\mathbb{A}_{11}^n$ and $\mathbb{A}_{11}^\zeta$ in PCG-V-ASP).

Operation	PCG-V [s]	PCG-V-ASP [s]	Speedup
CG (1 it.)	0.04	0.04	1.0
V (1 it.)	0.73	0.29	2.5
V-smoothing (1 it.)	0.13	0.13	1.0
V-lowest level (1 it.)	0.60	0.16	3.8
Solver (1 it.)	0.77	0.33	2.3
No. of iters. ($\epsilon < 1e^{-6}$)	27	101	0.3
Solver (all)	20.68	33.30	0.6
Acceleration over CPU	3.2	7.0	-

The relative performance of PCG-V-ASP with respect to PCG-V does however improve significantly if both solvers are accelerated using a GPU. Table 10 shows the performance of the PCG-V and PCG-V-ASP solvers, with most of the computations offloaded to a GPU (Table 3). The only part the computations that needs to be performed on a CPU is the direct solution of various systems of equations on the lowest level. This involves transferring the intermediate results between the GPU and CPU. Since PCG-V-ASP involves smaller matrices on the lowest level, the CPU-only part of the entire algorithm decreases significantly with respect to the standard PCG-V, and the impact using a GPU is greater for PCG-V-ASP than for PCG-V. This is clearly visible comparing the data in the fifth rows of Tables 9 and 10. For a CPU-only implementation, the operations on the lowest level are 1.9 faster for PCG-V-ASP, while this speedup increases to 3.8 for the implementation involving a GPU. If the CPU-only implementation is compared to the GPU version, it can be seen that the use of the GPU reduces the time taken by a single iteration from 2.48 to 0.77 (3.2 times) for the PCG-V and from 2.3 to 0.33s (a factor of seven). The overall impact of the GPU can be seen in the final row of the Table 10, which presents the speedup of the GPU-version over the CPU-only version, considering all operations.

TABLE 11. Comparison of the time taken to solve a complex-valued sparse system of equations with 5.1 million unknowns. LDL^T factorization of sparse matrices on the lowest level of V and V-ASP preconditioners was used.

Implementation	Device	Time [s]
PCG-V (reference)	CPU-based	66.9
PCG-V-ASP	CPU-based	232.5
PCG-V	GPU-based	20.7
PCG-V-ASP	GPU-based	33.3

Table 11 compares the time taken to solve a complex-valued sparse system of equations with 5.1 million unknowns for various implementations of preconditioned conjugate gradient solver. Since PCG-V-ASP requires over three times more iterations, the time taken by the PCG-V solver to achieve the assumed convergence is shorter. However, the GPU offsets, to a certain extent, the greater-than 3.74 to 1 difference in the number of iterations, reducing the runtime advantage of the PCG-V to a ratio of 2 to 1. Although ultimately the PCG-V-ASP solver is slower, the difference on a GPU is not very dramatic, and with the GPU acceleration, this preconditioner becomes a viable option for even larger complex-valued systems (with more than five million unknowns), for which the factorization of the matrix $\mathbb{A}_{11}$ cannot be handled due to the memory limitations of the workstation.

VI. CONCLUSION

In this paper, fast CPU-based and GPU-based implementations of the conjugate gradient iterative method, using parallel preconditioners with low memory requirements, were employed to solve complex-valued and sparse systems resulting from the FEM democratization of time-harmonic Maxwell's equations. Both preconditioners make use of the hierarchy of basis functions and operate in a V-cycle. In one preconditioner, a direct solution of the sparse system of equations is performed, while the other involves an iterative solution with an auxiliary space-preconditioning ASP. ASP has been advocated in the past as an approach that significantly reduces the memory footprint. This paper shows that, when symmetry is taken into account and Intel Pardiso LDL^T factorization using is used, the memory requirements improvements are small. Moreover, for the CPU-only implementation, the PCG-V-ASP solver turned out to be over

three times slower than the solver using a direct solution on the lowest level. It can be concluded that the use of ASP does not pay off on multicore CPUs. The use of a graphics processing unit (Pascal P100) results in a significant acceleration of computations (by a factor of seven in the PCG-V-ASP solver and by a factor of 3.2 for the PCG-V solver) compared to parallel CPU-only computations involving all CPU cores and using optimized vendor libraries. Even better speedups can be expected for problems with multiple right-hand sides, where blocking allows better utilization of the computational resources of the GPU [26]. Since the PCG-V-ASP solver benefits more from the GPU acceleration, it seems that that a blocked version of the PCG-V-ASP GPU-accelerated code could prove competitive compared to a blocked PCG-V solver.

REFERENCES

[1] H. A. van der Vorst and J. B. M. Melissen, "A Petrov-Galerkin type method for solving Axk = b, where A is symmetric complex," *IEEE Trans. Magn.*, vol. 26, no. 2, pp. 706–708, Mar. 1990.

[2] L. Li, T.-Z. Huang, and Z.-G. Ren, "A preconditioned COCG method for solving complex symmetric linear systems arising from scattering problems," *J. Electromagn. Waves Appl.*, vol. 22, nos. 14–15, pp. 2023–2034, 2008.

[3] X. M. Gu, T. Z. Huang, L. Li, H. B. Li, T. Sogabe, and M. Clemens, "Quasi-minimal residual variants of the COCG and COCR methods for complex symmetric linear systems in electromagnetic simulations," *IEEE Trans. Microw. Theory Techn.*, vol. 62, no. 12, pp. 2859–2867, Dec. 2014.

[4] Y. Saad and M. H. Schultz, "GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems," *SIAM J. Sci. Statist. Comput.*, vol. 7, no. 3, pp. 856–869, 1986.

[5] J. Gao, K. Wu, Y. Wang, P. Qi, and G. He, "GPU-accelerated preconditioned GMRES method for two-dimensional Maxwell's equations," *Int. J. Comput. Math.*, vol. 94, no. 10, pp. 2122–2144, 2017.

[6] Y. Zhu and A. C. Cangellaris, *Multigrid Finite Element Methods for Electromagnetic Field Modeling*. Hoboken, NJ, USA: Wiley, 2006, vol. 28.

[7] J.-F. Lee and D.-K. Sun, "P-type multiplicative Schwarz (pMUS) method with vector finite elements for modeling three-dimensional waveguide discontinuities," *IEEE Trans. Microw. Theory Techn.*, vol. 52, no. 3, pp. 864–870, Mar. 2004.

[8] Y. Zhu and A. C. Cangellaris, "Hierarchical multilevel potential preconditioner for fast finite-element analysis of microwave devices," *IEEE Trans. Microw. Theory Techn.*, vol. 50, no. 8, pp. 1984–1989, Aug. 2002.

[9] 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.

[10] 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.

[11] A. Aghabarati and J. P. Webb, "Multilevel methods for *p*-adaptive finite element analysis of electromagnetic scattering," *IEEE Trans. Antennas Propag.*, vol. 61, no. 11, pp. 5597–5606, Nov. 2013.

[12] J. Xu, "The auxiliary space method and optimal multigrid preconditioning techniques for unstructured grids," *Computing*, vol. 56, no. 3, pp. 215–235, 1996.

[13] A. Aghabarati and J. P. Webb, "An algebraic multigrid method for the finite element analysis of large scattering problems," *IEEE Trans. Antennas Propag.*, vol. 61, no. 2, pp. 809–817, Feb. 2013.

[14] A. Aghabarati and J. P. Webb, "Algebraic multigrid combined with domain decomposition for the finite element analysis of large scattering problems," *IEEE Trans. Antennas Propag.*, vol. 63, no. 1, pp. 404–408, Jan. 2015.

[15] T. A. Davis, "Algorithm 832: UMFPACK V4. 3—An unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 196–199, 2004.

[16] UMFPACK. *SuiteSparse: A Suite of Sparse Matrix Software*. Accessed: Sep. 9, 2017. [Online]. Available: http://faculty.cse.tamu.edu/davis/suitesparse.html

[17] O. Schenk and K. Gärtner, "Solving unsymmetric sparse systems of linear equations with PARDISO," *Future Gener. Comput. Syst.*, vol. 20, no. 3, pp. 475–487, 2004.

[18] 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.

[19] T. A. Davis, "A column pre-ordering strategy for the unsymmetric-pattern multifrontal method," *ACM Trans. Math. Softw.*, vol. 30, no. 2, pp. 165–195, 2004.

[20] 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.

[21] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2003.

[22] 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.

[23] R. Hiptmair and J. Xu, "Nodal auxiliary space preconditioning in *H* (curl) and *H* (div) spaces," *SIAM J. Numer. Anal.*, vol. 45, no. 6, pp. 2483–2509, 2007.

[24] J. Xu, L. Chen, and R. H. Nochetto, "Optimal multilevel methods for *H* (grad), *H* (curl), and *H* (div) systems on graded and unstructured grids," in *Multiscale, Nonlinear and Adaptive Approximation*. Berlin, Germany: Springer, 2009, pp. 599–659.

[25] 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, Apr. 2011.

[26] 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.

[27] F. Alessandri *et al.*, "The electric-field integral-equation method for the analysis and design of a class of rectangular cavity filters loaded by dielectric and metallic cylindrical pucks," *IEEE Trans. Microw. Theory Techn.*, vol. 52, no. 8, pp. 1790–1797, Aug. 2004.

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, respectively. He was also a recipient of the Prime Minister's Award for his Ph.D. thesis in 2016.

GRZEGORZ FOTYGA received the M.S.E.E. and Ph.D. degrees in electronic engineering from the Gdańsk University of Technology in 2009 and 2016, respectively. He is currently an Assistant Professor with the Department of Microwave and Antenna Engineering, Gdańsk University of Technology. His current research interests include computational electromagnetics, numerical methods, the finite-element method, and model-order reduction.

MICHAL MROZOWSKI (S'88–M'90–SM'02–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.