MPI-CUDA parallel linear solvers for block-tridiagonal matrices in the context of SLEPc's eigensolvers[☆]

A. Lamas Daviña, J. E. Roman[*]

D. Sistemes Informàtics i Computació, Universitat Politècnica de València, Camí de Vera s/n, 46022 València, Spain

Abstract

We consider the computation of a few eigenpairs of a generalized eigenvalue problem $Ax = \lambda Bx$ with block-tridiagonal matrices, not necessarily symmetric, in the context of Krylov methods. In this kind of computation, it is often necessary to solve a linear system of equations in each iteration of the eigensolver, for instance when B is not the identity matrix or when computing interior eigenvalues with the shift-and-invert spectral transformation. In this work, we aim to compare different direct linear solvers that can exploit the block-tridiagonal structure. Block cyclic reduction and the Spike algorithm are considered. A parallel implementation based on MPI is developed in the context of the SLEPc library. The use of GPU devices to accelerate local computations shows to be competitive for large block sizes.

Keywords: MPI, GPU computing, Eigenvalue computation, Block-tridiagonal linear solvers

1. Introduction

We are interested in computing a few eigenvalues (and corresponding eigenvectors) of a matrix (or matrix pair) that has block-tridiagonal struc-

[☆]This work was supported by Agencia Estatal de Investigación (AEI) under grant TIN2016-75985-P, which includes European Commission ERDF funds. Alejandro Lamas Daviña was supported by the Spanish Ministry of Education, Culture and Sport through a grant with reference FPU13-06655.

[*]Corresponding author.

Email addresses: alejandro.lamas@dsic.upv.es (A. Lamas Daviña), jroman@dsic.upv.es (J. E. Roman)

ture. We focus on the case where the blocks are dense, for instance in the discretization of partial differential equations in which the Fourier transform has been applied on two spatial dimensions [1].

Our particular interest is in assessing the convenience of using a combined MPI-CUDA parallelization scheme in order to benefit from graphics processing units (GPU) that are often available in computing clusters nowadays. This approach will perform best when addressing very large-scale problems, with many blocks of considerable size.

Given two $n \times n$ matrices A and B, we seek pairs (x, λ) satisfying the relation

$$Ax = \lambda Bx, \tag{1}$$

where λ is a scalar (eigenvalue) and $x \neq 0$ is a vector (eigenvector). This is called the generalized eigenvalue problem, while the particular case when B is the identity matrix is referred to as the standard eigenvalue problem. Our description will assume that the matrices are real, although our code can also handle the case of complex matrices. When the matrix pencil (A, B) is symmetric positive-definite, then all eigenvalues are real, otherwise eigenvalues are complex in general. We do not restrict ourselves to the symmetric case.

In this work, we consider Krylov methods for solving the eigenvalue problem, rather than direct methods. The main reason is that we need to compute just a few eigenpairs, not the whole spectrum. An additional reason is that, while the block-tridiagonal structure may seem favorable for transformation methods that reduce to a condensed form, this is not true for the non-symmetric case since already the first step (reduction to Hessenberg form) will destroy the block-tridiagonal structure. Even in the symmetric case, if the problem is generalized ($B \neq I$) then one has to first transform to a standard eigenproblem that also implies losing the block-tridiagonal structure. In contrast, Krylov methods will preserve the block-tridiagonal structure throughout the computation, and will be very effective provided that (1) convergence is not too bad, and (2) operations related to block-tridiagonal matrices are implemented efficiently.

Convergence of Krylov methods for eigenvalue computations is a non-trivial issue, that depends on separation of eigenvalues, among other things (see for instance [2]). Without entering into details, we could state that, if eigenvalues of interest are exterior (that is, lying in the periphery of the spectrum), a (restarted) Krylov method will be able to retrieve them without problems. However, if we seek for eigenvalues in the interior of the spectrum,

then it is necessary to do something else. A standard technique to compute interior eigenvalues is the shift-and-invert transformation, where the Krylov method is applied to the transformed problem

$$(A - \sigma B)^{-1} B x = \theta x, \tag{2}$$

where largest magnitude $\theta = (\lambda - \sigma)^{-1}$ correspond to eigenvalues λ closest to a given target value σ. Convergence in this case will be fast because the transformation also improves the separation, but the drawback is that the solver must implicitly handle the inverse of $A - \sigma B$ via direct linear solves. Direct linear solvers are costly because they need a matrix factorization, and scale quite poorly in parallel due to the required triangular solves. In this paper, we put special emphasis on efficient and scalable solution of linear systems in this context, exploiting the block-tridiagonal structure, in an MPI-CUDA programming model.

We now mention some related works, especially those dealing with the computation of eigenvalues using GPUs. Some authors have focused on increasing the arithmetic intensity for the first step of dense methods, namely the reduction to Hessenberg (or tridiagonal) form via orthogonal transformations [3, 4]. As pointed out before, these techniques are not relevant for our problem, because they would destroy the block-tridiagonal structure. Furthermore, we are interested in very large problems that do not fit the memory of a single GPU. In dense methods, after reducing to Hessenberg (or tridiagonal) form, the algorithm that actually computes the eigenvalues is then invoked, for instance the QR iteration for non-symmetric problems, or the divide-and-conquer method for symmetric ones. In contrast to the reduction step, this part is difficult to implement and has modest arithmetic intensity. Investigations to enhance efficiency have focused on using a hybrid CPU-GPU approach [5] or extending the algorithms to operate directly on a symmetric band matrix [6]. These methods compute all eigenvalues, which is wasteful in our case. With respect to projection methods for sparse matrices, recently Anzt *et al.* [7] have shown good performance results of a LOBPCG implementation for GPUs. Unfortunately, the LOBPCG eigensolver is specific for symmetric problems and cannot compute interior eigenvalues; we are targeting a much more general case. Several authors have reported about GPU implementation of Krylov eigensolvers. For instance, Aliaga *et al.* [8] accelerates Krylov methods by off-loading the matrix-vector products to the GPU, after carrying out a band reduction of the matrix (and losing the

sparse character). This work is restricted to computing exterior eigenvalues of symmetric matrices, while our main concern is interior eigenvalues of non-symmetric block-tridiagonal matrices.

The rest of the paper is organized as follows. Section 2 describes SLEPc, the library in which our codes are based, paying special attention to its support for GPUs. Section 3 describes different algorithms for the parallel solution of linear systems with block-tridiagonal matrices, and section 4 discusses specific details of our particular implementation of such algorithms. Performance results are reported in section 5. Finally, some concluding remarks are given in section 6.

2. Sparse linear algebra computations on GPU

Our code has been developed on top of SLEPc and PETSc, and we describe next how these libraries provide GPU support. We must note that, although some parts of the computation are carried out within these libraries, the majority of the computational cost lies in the user-provided subroutines, most notably those solving linear systems of equations that will be described in section 3.

SLEPc, the Scalable Library for Eigenvalue Problem Computations [9], is a parallel library intended to solve large-scale eigenvalue problems, computing a subset of eigenvalues and associated eigenvectors. It covers both the standard and generalized eigenproblems, as well as other related problems. SLEPc provides a collection of eigensolvers, including a restarted Krylov method which is the default one. It also furnishes a built-in tool for the shift-and-invert transformation (2) to compute interior eigenvalues.

SLEPc is built on top of PETSc[10], an object-oriented parallel framework for the numerical solution of partial differential equations. In PETSc the code is organized as a set of data structures for representing, e.g., vectors and matrices, and algorithmic objects (solvers) for different types of problems, including linear systems of equations. The application code interacts with the interface of these objects without caring about the details of the underlying data structures. In particular, SLEPc solvers are data-structure neutral, meaning that the computation can be done with different sparse matrix storage formats, for instance one that keeps a copy of the data on the GPU, as explained below. Furthermore, both SLEPc and PETSc are extensible in the sense that it is possible to plug user-provided code to customize

the solvers. We have used this feature to implement optimized kernels for block-tridiagonal matrices.

In the development version, PETSc incorporates some support for NVIDIA GPUs. The initial implementation [11] required an external package called CUSP [12], but during the development of this work we have contributed code to PETSc in order to avoid this dependence and enable GPU support simply with the NVIDIA CUDA toolkit[1]. Our contributions also allow some enhancements such as the use of complex arithmetic on the GPU.

The GPU support in PETSc is based on using cuBLAS and cuSPARSE[2] [13] to perform vector operations and matrix-vector products through VECCUDA, a special vector class whose array is mirrored in the GPU, and MATAIJCUSPARSE, a matrix class where data generated on the host is copied to the device on demand. The GPU model considered in PETSc uses MPI for communication between different processes, each of them having access to a single GPU [11]. The implementation includes mechanisms to guarantee coherence of the mirrored data-structures in the host and the device.

SLEPc inherits this GPU support philosophy, and extends it a bit with some multi-vector operations that are often encountered in eigenvalue methods. This operations replace BLAS-1 computations with BLAS-2 (and even BLAS-3 in some cases), with the consequent increase in arithmetic intensity. When computing eigenvalues in the periphery of the spectrum, it is possible to perform most of the computation on the GPU, see [14] for a sample application. However, when computing interior eigenvalues with the shift-and-invert transformation (2), a direct linear solver is required to handle the inverse, and this is not currently available in PETSc for GPU. Hence, in this case data would be copied back and forth between the CPU and the GPU, in a transparent way, with a considerable loss of efficiency. Sparse direct solvers are difficult to implement on the GPU efficiently. Here, we restrict to the simpler case of block-tridiagonal matrices, and have implemented custom subroutines that work completely on the GPU. In a previous work [15], we presented some results for the case of a single GPU, and here we consider the general case where both MPI and GPU are used in the parallelization of

[1]PETSc also features OpenCL support for GPU hardware other than NVIDIA, but we do not consider this here.

[2]cuBLAS and cuSPARSE are libraries included in the CUDA software development toolkit: cuBLAS implements BLAS for CUDA, while cuSPARSE provides linear algebra operations on sparse matrices.

linear solves.

The default eigensolver in SLEPc is the Krylov-Schur method [16], which essentially consists in the Arnoldi recurrence enriched with an effective restart mechanism. After $j-1$ steps, the Arnoldi recurrence computes an orthogonal basis of the Krylov subspace $\mathcal{K}_j(M, v_1) := \text{span}\{v_1, Mv_1, M^2v_1, \ldots, M^{j-1}v_1\}$, where v_1 is a unit-norm initial vector. In the generalized eigenvalue problem (1), the subspace is built for matrix $M = B^{-1}A$ (assuming B is non-singular), or alternatively $M = (A - \sigma B)^{-1}B$ if the shift-and-invert transformation (2) is being used.

In some cases, Ritz approximations may require many iterations to converge. The number of Arnoldi steps cannot be arbitrarily large, because of storage requirements for the basis and also because the computational cost grows with j. For this reason, a practical implementation must be able to *restart*, that is, discard part of the information contained in the subspace and extend the subspace again. In Krylov-Schur [16], restart is accomplished by truncating the Krylov decomposition in a simple-to-implement way.

Without entering into the details of the algorithm, the different steps involve the following computational kernels:

1. Basis expansion. The last Arnoldi vector v_j must be multiplied by M. In the considered cases, this is a matrix-vector multiplication, followed by a linear system solve.
2. Orthogonalization and normalization of vectors. The new vector must be orthonormalized against the previous ones, $V_j = [v_1, v_2, \ldots, v_j]$.
3. Solution of projected eigenproblem. A small eigenvalue problem must be solved at each restart, for matrix $H_j = V_j^* M V_j$.
4. Restart. The associated computation is $V_j Q_{1:k}$, $k < j$, where Q contains the Schur vectors of H_j.

The cost of step 3 is usually negligible compared to the rest of operations, because we are assuming that the size of the projected problem is very small (often several orders of magnitude smaller) with respect to the original problem size. Hence this computation is not parallelized, it is performed in a replicated way on the CPU for every MPI process. Regarding steps 2 and 4, they involve vector operations that are easily parallelizable in terms of MPI. They can also be mapped easily to the GPU since they involve simple calls to the BLAS. Finally, the basis expansion (step 1) is the most critical computation, since it is often the most expensive one and may be difficult to implement efficiently in parallel, especially in cases where it

involves solving linear systems. In the next section we provide details for this step, particularized for the case of block-tridiagonal matrices.

3. Parallel linear system solvers for block-tridiagonal matrices

As discussed above, in Krylov methods the basis expansion with matrix M involves a matrix-vector multiplication and a linear system solve with either A, B, or $A-\sigma B$, depending on whether shift-and-invert is being used or not. In this paper, we are assuming that all these matrices have a block-tridiagonal structure.

Consider a block-tridiagonal matrix T of order n, with ℓ blocks of size k,

$$T = \begin{bmatrix} B_1 & C_1 & & & & \\ A_2 & B_2 & C_2 & & & \\ & A_3 & B_3 & C_3 & & \\ & & \ddots & \ddots & \ddots & \\ & & & A_{\ell-1} & B_{\ell-1} & C_{\ell-1} \\ & & & & A_\ell & B_\ell \end{bmatrix}, \tag{3}$$

with $n = \ell k$. The matrix-vector product $y = Tv$ can be easily computed by blocks as

$$y_i = [A_i \; B_i \; C_i] \begin{bmatrix} v_{i-1} \\ v_i \\ v_{i+1} \end{bmatrix}, \qquad i = 2, \ldots, \ell - 1, \tag{4}$$

with analogous expressions for the first and last block-rows. The vectors y and v have been partitioned in a way consistent with T. As will be discussed in section 4, we choose to store the matrix in such a way that it is possible to do the computation with a single call to BLAS _gemv per each block-row (or the equivalent cuBLAS subroutine for the GPU case). In the MPI setting, the matrix is distributed by blocks of block-rows, where none of the ℓ blocks are cut across process boundaries. Then, the multiplication with the local matrix must be preceded by the exchange of one block of v between neighbouring processes.

In contrast to the matrix-vector product operation, the solution of linear systems is much more difficult to implement in an efficient way. The rest of this section is devoted to describing the algorithms that we have implemented for this.

3.1. Direct linear solvers

We now focus on direct methods to solve linear systems of the form

$$Tx = z. \tag{5}$$

As can be found in [17, ch. 5], besides the classical Gaussian elimination there are several well known algorithms to solve tridiagonal linear systems such as recursive doubling, cyclic reduction or parallel cyclic reduction. These methods can also be extended to the block-tridiagonal case. One of the solvers that we have implemented is based on the cyclic reduction [18] (also known as odd/even reduction or CORF).

The cyclic reduction algorithm has two main steps: the forward elimination, that reduces the rows of the system, and the backward substitution to obtain the solution. It is a recursive algorithm that successively reduces the number of rows. It divides the rows in even-indexed and odd-indexed, and in the forward elimination, it recursively eliminates the even-indexed rows in terms of the odd-indexed ones. Depending on the dimension of the matrix, the number of rows is approximately halved in each recursion (if the number of rows is a power of two, it is actually halved) until a single row is left. Assuming that no division by zero is encountered in any of these steps, with the last row, an equation with a single unknown is trivially solved. The backward substitution step progresses increasing the number of rows used in the same proportion as the forward elimination reduces them. It uses the current recursion level solution(s) to compute the adjacent even-indexed rows on the previous level until all the unknowns are solved.

As noted in [19, 20], the cyclic reduction method has the property of being equivalent to Gaussian elimination without pivoting on the system $(PTP^T)(Px) = Pz$, where P is a permutation matrix that places first indices that are odd multiples of 2^0, then odd multiples of 2^1, and so on. Such link with Gaussian elimination supports the conclusion that cyclic reduction is numerically stable in the same cases where Gaussian elimination with diagonal pivots is. If T is strictly diagonally dominant or symmetric positive definite, then no pivoting is necessary and cyclic reduction is stable.

A version of the algorithm that works with general block-tridiagonal matrices was proposed in [21] and more recently reworked in [1] as BCYCLIC, where block-rows are reduced cyclically instead of rows. In this case, the algorithm can progress as long as the diagonal blocks are non-singular, since it is necessary to compute their inverse, or, more precisely, their inverse is

handled implicitly via factorization. Pivoting can be used when factorizing the diagonal blocks but this does not guarantee the stability of the algorithm. Some aspects of the numerical stability of the block cyclic reduction were analyzed in [21] and [22], where a study of the bounds of the forward error is conducted for the cases of (strictly) diagonally dominant matrices assuming that the matrix is block column diagonal dominant.

During the forward elimination stage, the block cyclic reduction computes the inverse of the even-indexed diagonal blocks and a modified (hatted) version of the lower and upper blocks. In the same way, a modified version of the even-indexed blocks of the right-hand side (RHS) vector z is computed. In the first recursion, the computed quantities are

$$\hat{B}_{2j} = B_{2j}^{-1}, \tag{6a}$$
$$\hat{A}_{2j} = \hat{B}_{2j} A_{2j}, \tag{6b}$$
$$\hat{C}_{2j} = \hat{B}_{2j} C_{2j}, \tag{6c}$$
$$\hat{z}_{2j} = \hat{B}_{2j} z_{2j}, \tag{6d}$$

for $j = 1, \ldots, \ell/2$. The respective modified version of the odd-indexed blocks is then computed by using the adjacent modified even-indexed blocks,

$$\hat{B}_{2j-1} = B_{2j-1} - A_{2j-1}\hat{C}_{2j-2} - C_{2j-1}\hat{A}_{2j}, \tag{7a}$$
$$\hat{A}_{2j-1} = -A_{2j-1}\hat{A}_{2j-2}, \tag{7b}$$
$$\hat{C}_{2j-1} = -C_{2j-1}\hat{C}_{2j}, \tag{7c}$$
$$\hat{z}_{2j-1} = z_{2j-1} - A_{2j-1}\hat{z}_{2j-2} - C_{2j-1}\hat{z}_{2j}. \tag{7d}$$

In subsequent recursion levels, the computation is analogous to (6)-(7), but for a matrix that has about half of the blocks with respect to the previous level (even blocks have been removed).

In an MPI implementation, if the matrix is distributed across several processes, prior to the computation of (7) it is necessary to send $\hat{A}_{2j}$, $\hat{C}_{2j}$ and $\hat{z}_{2j}$ to the processes that own the adjacent odd-indexed block-rows, so communication is necessary in this case. That occurs for every recursive step of the algorithm. The modified even-indexed blocks can overwrite the original ones, but for the odd-indexed blocks, in order to back-solve with multiple right-hand sides, both the original lower and upper blocks and their modified versions must be retained producing a 66% increase of memory usage during this stage.

The backward substitution stage starts by solving the single block equation

$$x_1 = \hat{B}_1 \hat{z}_1, \tag{8}$$

where $\hat{B}_1$ and $\hat{z}_1$ correspond to quantities computed in the last recursion level. Once this final odd-indexed block, which is the first part of the solution, is obtained, the recursion tree is traversed in reverse order. The solution blocks from a certain recursion level are used to compute the adjacent even-indexed blocks on the previous level, with

$$x_{2j} = \hat{z}_{2j} - \hat{A}_{2j} x_{2j-1} - \hat{C}_{2j} x_{2j+1}. \tag{9}$$

Communication occurs in an analogous way as in the forward elimination, but in the opposite direction. The algorithm continues until all blocks are processed.

The computational cost is concentrated in the first 3 operations of (6) and (7). These operations represent the factorization itself, and can be amortized in the case of multiple right-hand sides. This is what happens in the Arnoldi method, that needs to invoke the linear solver in each iteration of the eigensolver.

An alternative algorithm that uses a different approach is Spike [23], which is intended for the parallel solution of banded linear systems (with a possibly sparse band). Here, we particularize the algorithm for our block-tridiagonal structure. At the outset of the algorithm, matrix T is partitioned and distributed evenly among the p available processes, and, as before, we assume that none of the ℓ block-rows are split across different processes. Each process r organizes its local data in the form

$$T_r = \left[\begin{array}{c|c|c|c|c} 0 & \begin{matrix} D_r \\ 0 \end{matrix} & E_r & \begin{matrix} 0 \\ F_r \end{matrix} & 0 \end{array} \right], \quad r = 0, \ldots, p-1, \tag{10}$$

distinguishing between the diagonal portion E_r (that is, the column range corresponding to local rows), and the lower (D_r) and upper (F_r) blocks that represent the coupling with neighbouring processes. Note that the sizes of these three blocks differ, being the diagonal block larger than the lower and upper blocks. In our case, the size of D_r and F_r is k, the original block size of the block-tridiagonal matrix, whereas the diagonal block has an approximate size of n/p, being p the number of processes used.

The Spike algorithm has two main phases: factorization and post-processing. The factorization phase consists in computing the so-called *Spike* matrix, S,

defined from the decomposition $T = ES$, where $E = \mathrm{diag}(E_0, E_1, \ldots, E_{p-1})$. This amounts to multiplying the local matrix T_r by the inverse of the diagonal block E_r to get the local matrix S_r. In order to do that, the first step is to factorize the diagonal blocks E_r, and compute matrices V_r and W_r by solving the system

$$E_r \begin{bmatrix} V_r & W_r \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} 0 \\ F_r \end{bmatrix} & \begin{bmatrix} D_r \\ 0 \end{bmatrix} \end{bmatrix} \tag{11}$$

on each process. We must emphasize that (11) is a system of linear equations with $2k$ right-hand sides, where the coefficient matrix is block-tridiagonal, and hence a solver that exploits this structure can be used. Furthermore, this computation can be performed independently by each process, without communication.

These V_r and W_r matrices are the spikes on a matrix S whose main block-diagonal is the identity matrix,

$$S = \begin{bmatrix} I & \hat{V}_0 & & & & \\ \hat{W}_1 & I & \hat{V}_1 & & & \\ & \hat{W}_2 & I & \hat{V}_2 & & \\ & & \ddots & \ddots & \ddots & \\ & & & \hat{W}_{p-2} & I & \hat{V}_{p-2} \\ & & & & \hat{W}_{p-1} & I \end{bmatrix}, \tag{12}$$

where we use the notation $\hat{V}_r = \begin{bmatrix} V_r & 0 \end{bmatrix}$ and $\hat{W}_r = \begin{bmatrix} 0 & W_r \end{bmatrix}$.

For the post-processing stage, each process divides the local portion of the matrix S in top, middle and bottom parts, being the top and bottom parts of size k, and the middle part of the same size as E_r minus $2k$. For instance, the upper spike is written as $V_r = \begin{bmatrix} V_r^{(t)} \\ V_r^{(m)} \\ V_r^{(b)} \end{bmatrix}$. Selecting the top and

bottom blocks of the spikes, a reduced matrix

$$\hat{S} = \begin{bmatrix} I & & V_0^{(t)} & & & & & \\ & I & V_0^{(b)} & & & & & \\ & W_1^{(t)} & I & & V_1^{(t)} & & & \\ & W_1^{(b)} & & I & V_1^{(b)} & & & \\ & & \ddots & \ddots & \ddots & & & \\ & & & W_{p-2}^{(t)} & I & & V_{p-2}^{(t)} & \\ & & & W_{p-2}^{(b)} & & I & V_{p-2}^{(b)} & \\ & & & & & W_{p-1}^{(t)} & I & \\ & & & & & W_{p-1}^{(b)} & & I \end{bmatrix} \tag{13}$$

is built, also with an identity matrix on its diagonal.

The first stage in the post-processing phase is the factorization of the reduced matrix $\hat{S}$. For the algorithm to work, the size of the original matrix has to be large enough to distribute at least two blocks per process, so the number of processes has to be limited to allow this ratio when factorizing small matrices.

Once $\hat{S}$ has been factored, the next stage is to locally compute vector $g = E^{-1}z$ by solving the system

$$E_r g_r = z_r, \tag{14}$$

in a similar way as (11). Note that z_r now denotes the whole subvector of z stored locally in process r, not an individual block. Vector g_r is in turn partitioned in the same top, middle and bottom parts as the spikes. From the top and bottom parts of each process, a reduced vector $\hat{g}$ is formed and used as RHS to solve the reduced system

$$\hat{S}\hat{x} = \hat{g}, \tag{15}$$

where $\hat{x}$ refers to the top and bottom blocks of x on each process.

Finally, to fully solve the whole system, the middle parts of x can be obtained in parallel with

$$\begin{cases} x_0^{(m)} = g_0^{(m)} - V_0^{(m)}\hat{x}_1^{(t)}, \\ x_j^{(m)} = g_j^{(m)} - V_j^{(m)}\hat{x}_{j+1}^{(t)} - W_j^{(m)}\hat{x}_{j-1}^{(b)}, \qquad j = 1, \ldots, p-2, \\ x_{p-1}^{(m)} = g_{p-1}^{(m)} - W_{p-1}^{(m)}\hat{x}_{p-2}^{(b)}. \end{cases} \tag{16}$$

For the case of diagonally dominant matrices, as noted in [23], it is interesting to consider that the value of the elements of the right spikes (V_r) decays in magnitude from bottom to top. The decay is greater, the more diagonally dominant the matrix is. As a consequence, the value of the elements of the top block of V_r can be expected to be zero and discarded from the computation. The same assumption can be done for the bottom block of the left spikes W_r. Discarding these zero blocks, the extra reduced matrix

$$\bar{S} = \begin{bmatrix} I & V_0^{(b)} & & & & & & & \\ W_1^{(t)} & I & & & & & & & \\ & & I & V_1^{(b)} & & & & & \\ & & W_2^{(t)} & I & & & & & \\ & & \ddots & \ddots & \ddots & & & & \\ & & & & I & V_{p-3}^{(b)} & & & \\ & & & & W_{p-2}^{(t)} & I & & & \\ & & & & & & I & V_{p-2}^{(b)} \\ & & & & & & W_{p-1}^{(t)} & I \end{bmatrix} \tag{17}$$

can be built and factored as a block-diagonal matrix, instead of as a block-tridiagonal one. Data movement between processes is necessary to form the matrix $\bar{S}$, and all but one of the processes can work in parallel during the factorization without communication.

Polizzi and Sameh presented in [23] an optimization for the diagonally dominant case denoted as Truncated Spike that computes only the bottom block of V_r and the top block of W_r. It requires to perform one LU factorization and one UL factorization of the diagonal block E_r. Their experiments proved that this approach reduces the computing time on machines with arithmetic operations much faster than accesses to the memory. As the spikes are not fully computed, the post-processing stage differs from the general Spike algorithm. A study of the Truncated Spike [24] found that if the degree of diagonal dominance

$$d = \left(\max_i \left\{ \frac{\sum_{j \neq i} |a_{ij}|}{|a_{ii}|} \right\} \right)^{-1} \tag{18}$$

of the original matrix is not too close to 1 and if the partitions are sufficiently large, then the errors at every stage of the algorithm are small. A more recent work [25] discusses several strategies to approach the diagonally and non-diagonally dominant cases.

3.2. Inexact shift-and-invert

For the non-diagonally dominant case, the Truncated Spike can be used as a preconditioner of an iterative method to solve the linear system. In this case, the decay of the values in the spikes is considerably less prominent, and the Truncated Spike discards blocks with non zero elements. The larger the block size and/or number of processes are, the greater the valid data discarded on this step, and the worse the approximated solution is. In such case, a large number of outer iterations is needed, dramatically increasing the communication and the computation, making this algorithm not worthwhile to solve such kind of systems. We have not considered this variant in the numerical experiments of section 5.

4. Implementations

Given the block-tridiagonal matrix T of (3), we store it in memory as

$$\text{store}(T) = \begin{bmatrix} \square & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \\ \vdots & \vdots & \vdots \\ A_\ell & B_\ell & \square \end{bmatrix}, \tag{19}$$

where the $\square$ symbols indicate blocks with memory allocated but not being used. In the MPI implementation, block-rows are split across processes, and hence only the first and last process have an unused block. Regarding the GPU implementation, this compact storage shape allows us to use 2D memory (pitched memory) in CUDA where we store the beginning of the blocks aligned, which is important for coalesced memory accesses. As mentioned in the previous section, this storage allows us to use a single _gemv BLAS call per block-row to perform the matrix-vector product on CPU and GPU, but apart from this and for the GPU case, we have also implemented a customized CUDA kernel that performs the whole matrix-vector computation with a single kernel invocation.

As a classical algorithm, the cyclic reduction has been already widely implemented for different programming paradigms and computer architectures. Studies of its performance on GPU against other solvers for tridiagonal matrices were carried out by Zhang *et al.* [26]. MPI-based implementations were studied for the block-tridiagonal case in [1] and [27]. The authors of

the former combined the use of MPI parallelism over the block-rows with a threaded parallelism with OpenMP or GotoBLAS to perform the local operations. The effect of the block size in the performance was studied in [27]. A heterogeneous approach was implemented by Park and Perumalla [28], who use MPI and the block arithmetic is done simultaneously on GPU with cuBLAS or MAGMA [29] and multicore processors with ACML. Another single-GPU implementation for the block-tridiagonal case was done in [30], whose authors tested a variety of block and matrix sizes, showing that better performance is obtained with systems with relatively large block sizes by better utilizing the available GPU threads. Recently, a comparison of the classical solvers, including the Thomas algorithm (a specialized Gaussian elimination for tridiagonal systems), was addressed in [31], implementing them on CPU, many integrated cores (MIC) architecture and GPU accelerators for the case of using a single node.

The Spike algorithm is available in the Intel Spike library[3] with an MPI-based implementation. Another implementation, specific for tridiagonal matrices, was developed in [32] for the (multi-)GPU case and later included in the cuSPARSE library. A Spike variation, that makes use of QR factorization without pivoting via Givens rotations, presented in [33] as g-Spike, safeguards the algorithm in case that the partitioning of the matrix results in at least one of the diagonal blocks being singular. This work was later adapted to the Intel Xeon Phi platform in [34]. An implementation of the Truncated Spike was used as a preconditioner to solve tridiagonal linear systems on GPU in [35] through the use of the CUSP library.

In our case, we focus on developing fast implementations of the solvers making use of several NVIDIA GPUs to perform the block arithmetic managed by a multi-process MPI solution in a similar way as [28]. When using the Spike algorithm to solve a block-tridiagonal system, the band has to be large enough to cover all the entries of the original matrix and additional triangular zero borders are included in the computation. We avoid the extra work that this would entail by factorizing the main diagonal block exploiting its block-tridiagonal structure. We have implemented several versions of the algorithms, both in CPU and GPU, to compare their performance.

The implementations have been split in two subroutines, factorization and solve, as the factorization is invoked only once, and the solve is invoked

[3]https://software.intel.com/en-us/articles/intel-adaptive-spike-based-solver/

in each iteration of the Arnoldi algorithm. In both cases, BCYCLIC and Spike, the factorization of the matrix excludes the operations with the RHS vector. Those steps of the algorithms are done during the solve subroutine.

Algorithms 1 and 2 summarize the operations done on each of the subroutines of the BCYCLIC algorithm in an iterative implementation. Please note that some (parts) of the operations of the algorithms only take place if the involved blocks exist on the process at the current level. In Algorithm 1, those operations correspond to steps 11, 12, 14, 15 and 16, and in Algorithm 2 they correspond to steps 10, 12, 14, 21 and 22.

Step 2 of Algorithms 1 and 2, which is detailed in Algorithm 3, obtains the lower and upper limit of the range of block-rows owned by the calling process. Algorithm 1 goes exclusively through the $\lceil \log_2(\ell) \rceil$ levels of the forward elimination, while Algorithm 2 performs the forward elimination and the backward substitution.

Algorithm 1: BCYCLIC factorization

```
1  For all processes (with rank r) do in parallel
2      [low, high] = GetOwnershipRange(p,r,ℓ)
3      for i = 1 : ⌈log2(ℓ)⌉ do          /* For each iteration level */
4          begin = 2^(i-1) + 1
5          if begin ≤ high then
6              while begin < low do  begin = begin + 2^i
7              begin = begin − low
8          for j = begin : 2^i : ℓ_r do     /* For each even block row */
9              B̂_2j = B_2j^-1
10             Â_2j = B̂_2j A_2j
11             Ĉ_2j = B̂_2j C_2j
12         Send/receive Â and Ĉ to/from adjacent block-rows
13         for j = 1 : 2^i : ℓ_r do          /* For each odd block row */
14             Â_2j-1 = −A_2j-1 Â_2j-2
15             B̂_2j-1 = B_2j-1 − A_2j-1 Ĉ_2j-2 − C_2j-1 Â_2j
16             Ĉ_2j-1 = −C_2j-1 Ĉ_2j
17     if r == 0 then B̂_1 = B_1^-1
```

Algorithm 2: BCYCLIC solve

For all processes (with rank r) **do in parallel**
  [low, high] = GetOwnershipRange(r,ℓ,p)
  for $i = 1 : \lceil \log_2(\ell) \rceil$ **do** `/* For each iteration level */`
    begin $= 2^{i-1} + 1$
    if begin $\leq$ high **then**
      while begin $<$ low **do** begin = begin $+ 2^i$
      begin = begin $-$ low
    for j = begin $: 2^i : \ell_r$ **do** `/* For each even block row */`
      $\hat{z}_{2j} = \hat{B}_{2j} z_{2j}$
    Send/receive $\hat{z}$ to/from adjacent block-rows
    for $j = 1 : 2^i : \ell_r$ **do** `/* For each odd block row */`
      $\hat{z}_{2j-1} = z_{2j-1} - A_{2j-1}\hat{z}_{2j-2} - C_{2j-1}\hat{z}_{2j}$
  if $r == 0$ **then** $x_1 = \hat{z}_1 = \hat{B}_1 z_1$
  Send/receive x_1 to/from adjacent block-rows
  for $i = \lceil \log_2(\ell) \rceil : -1 : 1$ **do** `/* For each iteration level */`
    begin $= 2^{i-1} + 1$;
    if begin $\leq$ high **then**
      while begin $<$ low **do** begin = begin $+ 2^i$
      begin = begin $-$ low
    for i = begin $: 2^i : \ell_r$ **do** `/* For each even block row */`
      $x_{2j} = \hat{z}_{2j} - \hat{A}_{2j} x_{2j-1} - \hat{C}_{2j} x_{2j+1}$
    Send/receive x to/from adjacent block-rows

Algorithm 3: GetOwnershipRange

Input: number of processes: p, process identifier: r, number of block-rows: ℓ

Output: global index of the first block-row owned by the process: low, global index of the last block-row owned by the process: high

```
1 if r < (ℓ mod p) then
2     low = r(⌊ℓ/p⌋ + 2)
3     high = low + ⌊ℓ/p⌋ + 1
4 else
5     low = r⌊ℓ/p⌋ + (ℓ mod p) + 1
6     high = low + ⌊ℓ/p⌋
```

In Algorithm 1, there are two alternatives available to deal with the inverse of the diagonal blocks in step 9. One is to explicitly compute the inverse by means of LAPACK routines _getrf and _getri. It could seem inappropriate to do this due to the high cost of computing the inverse, but once it is computed, the rest of the steps of the algorithm can be done with optimized matrix-matrix and matrix-vector multiplications. The other alternative is to solve linear systems by using LAPACK routines _getrf and _getrs, that a priori seems the more reasonable way to go due to the cheaper cost of the operations.

In steps 14 and 22 of Algorithm 2, parts of the solution vector are exchanged between processes. One block of the solution vector can be adjacent to multiple blocks at different levels of the backward substitution, and those blocks can belong to the same or different processes. To avoid repeating the same communication twice between the same pair of processes, the implementations of Algorithm 2 store and take note of the parts of the solution vector already sent/received.

The mathematical library used within the CPU version is Intel's MKL (with threads enabled), but for the GPU version we have different implementations that use the cuBLAS or MAGMA libraries, respectively. Both GPU libraries provide batched operations allowing us to use a single function call to factorize, solve, invert or multiply all the even-indexed diagonal blocks owned by a process, per recursive step. We use these batched operations in the GPU implementations, but, in the case of the _gemm function, the

Table 1: BCYCLIC implementations with each of the mathematical libraries.

		Non-batched		Batched		
	Math library	_getri	_getrs	_getri_1	_getri_2	_getrs
CPU	MKL	✓	✓	-	-	-
GPU	cuBLAS	-	-	✓	✓	✓
	MAGMA	✓	✓	✓	✓	✓

batched variant is expected to perform well with small matrices only. In our case, we have created two variants of the GPU version that computes the inverse of the diagonal blocks, one with normal _gemm in a loop (denoted as _getri_1) and another one with batched _gemm (denoted as _getri_2). The CPU-equivalent implementations (run on GPU without batched operations) make use of the MAGMA library. Table 1 summarizes the different BCYCLIC implementations.

The implementations of the Spike algorithm make use of the BCYCLIC implementations to solve the linear systems involved as, in our case, the systems are block-tridiagonal. The factorization subroutine of Spike is represented in Algorithm 4, in which only step 5 requires communication.

Algorithm 4: Spike factorization

1 **For all** processes (with rank r) **do in parallel**
2 $\quad \tilde{E}_r = \text{bcyclic\_factorize}(E_r)$
3 $\quad [V_r, W_r] = \text{bc4spike\_solve}(\tilde{E}_r, F_r, D_r)$
4 $\quad$ Build $\hat{S}$ with top and bottom parts of V_r and W_r
5 $\quad \tilde{S} = \text{bcyclic\_factorize}(\hat{S})$

Step 2 factorizes the diagonal block (block-tridiagonal) E_r and step 3 solves the corresponding system (11) to compute the spikes. For step 3, a different version of Algorithm 2 was implemented to deal with multiple RHS and to send/receive multiple times the same block of the solution vector (in this case, the re-sending option was chosen to reduce the allocated memory by eliminating the buffer mechanism, as no communication exists on this step). Also, due to the high number of zeros in the RHS matrices

of (11), another modification was introduced in the BCYCLIC solve used (bc4spike_solve) that saves time by skipping operations involving zero blocks on every recursion of the algorithm.

The second factorization needed in step 5 of Algorithm 4 is the factorization of the reduced matrix ($\hat{S}$), that is also done by means of BCYCLIC and, in this case, communication occurs between the processes as the matrix is distributed across them.

The solve subroutine of Spike is shown in Algorithm 5 in which the steps involving communication are 4, 5 and 6. Step 8 of Algorithm 5, that computes the middle part of the final solution on each process, corresponds to (16).

Algorithm 5: Spike solve

1 **For all** processes (with rank r) **do in parallel**
2 $g_r = \text{bcyclic\_solve}(\tilde{E}_r, z_r)$
3 Build $\hat{g}$ with top and bottom parts of g_r
4 $\hat{x} = \text{bcyclic\_solve}(\tilde{S}, \hat{g})$
5 Send $\hat{x}_r^{(t)}$ and $\hat{x}_r^{(b)}$ to $r-1$ and $r+1$, respectively
6 Receive $\hat{x}_{r+1}^{(t)}$ and $\hat{x}_{r-1}^{(b)}$
7 $x^{(t)} = \hat{x}^{(t)}$
8 $x_r^{(m)} = g_r^{(m)} - V_r^{(m)}\hat{x}_{r+1}^{(t)} - W_r^{(m)}\hat{x}_{r-1}^{(b)}$
9 $x^{(b)} = \hat{x}^{(b)}$

The memory requirements of our Spike implementations are higher than those of BCYCLIC, since in addition to the original matrix size and the 66% extra amount of BCYCLIC to perform the factorization of the local diagonal block E_r, more memory has to be allocated to build the reduced matrix, whose block size is twice as large as the original block size, and it has its own 66% increase. Besides that, the auxiliary memory buffers used to send and receive blocks have to be of this new double block size.

For diagonally dominant matrices, a variant of the Truncated Spike algorithm that works with the extra reduced matrix $\bar{S}$ of (17) is used. In the sequel, this method is referred to as reduced Spike. In our case, the implementation does not limit the solve to the first and last $k \times k$ blocks, but computes the full spikes (done via the BCYCLIC algorithm). The reduction in the computation is obtained through the use of the extra reduced system, as the blocks discarded should not contain any nonzero elements. Since the

Table 2: Operations used to factorize the two matrices and solve the systems on the Spike implementations.

	Spike		Reduced Spike	
	Matrix E_r	Matrix $\hat{S}$	Matrix E_r	Matrix $\bar{S}$
Factorization subroutine	Fact. bcyclic Solve bc4spike	Fact. bcyclic	Fact. bcyclic Solve bc4spike	Fact. _getrf
Solve subroutine	Solve bcyclic	Solve bcyclic	Solve bcyclic	Solve _getrs

spikes are fully computed, the logic of the algorithm that processes the diagonal blocks E_r does not change with respect to the general Spike. The extra reduced matrix can be factored in parallel with a _getrf call on each process without any communication or additional storage.

The solve subroutine of both variants of the Spike algorithm differs in the second solve used with the reduced matrix. A summary of the different operations employed by both variants can be seen in Table 2.

5. Computational experiments

In this section, we present results of some computational experiments aiming at assessing the performance of our codes. We are especially interested in scalability for increasing number of MPI processes (and GPUs), and also in how the block size has an impact on performance. Another question we want to answer is which of the implemented variants performs best.

We point out that our codes are prepared for real and complex arithmetic, with both single and double precision. All presented results correspond to real scalars with double precision.

5.1. Computing platform

All executions have been carried out on a cluster equipped with four GPUs per node. The cluster is formed by 39 servers with two Intel Xeon E5-2630 v3 processors and 128 GB of RAM, interconnected with fourteen

data rate (FDR) Infiniband cards at 56 Gb/s in a switched fabric network topology, and two NVIDIA K80 cards (with two GPUs each).

The servers run RedHat Linux 6.7 as operating system, and our software has been compiled with gcc 4.6.1 using SLEPc and PETSc 3.7-dev, and linked with the Intel MKL 11.3.2, NVIDIA CUDA 7.5 and MAGMA 1.7.0 libraries. The MPI version used for the inter-process communication is the cluster's manufacturer bullxmpi 1.2.9.1.

Since our codes use a single GPU per process, the number of processes per node has been limited to four in order to fully utilize the servers without oversubscribing the GPU cards with more than one process when running the GPU executions. In the case of the CPU runs, the same limit of four processes per node has been used to have the same communication overhead, and in this case the number of threads has been set also to four, to allow the software to use all the computational cores (one thread per core) with the four processes.

5.2. Test cases

The ultimate goal of this work is to extend SLEPc with eigensolvers that specialize for the particular case of block-tridiagonal matrices, that for instance can appear in applications like [1], whose blocks are dense, and that can also be used for dense banded matrices generated by applications like [36].

For the scalability studies, we consider an application coming from astrophysics, where the matrices are banded (with a dense band) and we are able to generate any matrix size with arbitrary bandwidth. The integral operator $T : X \to X$, arising from a transfer problem in stellar atmospheres [36], is defined by

$$(T\varphi)(\tau) = \frac{\varpi}{2} \int_0^{\tau^\star} \int_1^\infty \frac{e^{-|\tau-\tau'|\mu}}{\mu} d\mu\varphi(\tau')d\tau, \quad \tau \in [0, \tau^\star], \tag{20}$$

which depends on the albedo, $\varpi \in [0, 1]$, and the optical thickness of the stellar atmosphere, $\tau^\star$. We are interested in the eigenvalue problem $T\varphi = \lambda\varphi$ with $\lambda \in \mathbb{C}$ and $\varphi \in X$. This problem can be solved via discretization, that is, by projection onto a finite dimensional subspace X_n, resulting in an algebraic eigenvalue problem $A_n x_n = \theta_n x_n$ of dimension n, where A_n is the restriction of the projected operator to X_n. Further details can be found in [37].

Due to the exponential decay, the matrix A_n has a banded structure, with a bandwidth depending on the ratio between the matrix size, n, and

Table 3: Block sizes and number of block-rows used for the strong and weak scaling experiments. The column of the weak scaling shows the number of rows used with 128 processes, that is halved with the number of processes.

	Number of block-rows	
Block size	Strong scaling	Weak scaling
64	4800	51200
96	3200	38400
128	2400	25600
256	1200	12800
384	800	9600
512	600	6400
640	480	5600
768	400	4800
896	343	4000
1024	300	3200

the parameter $\tau^\star$,

$$b_w = \left\lfloor n \left(1 - \exp\left(-\frac{n}{t_c \tau^\star} \right) \right) \right\rfloor, \tag{21}$$

where $t_c = \max(n/100, 5)$. We always choose $\tau^\star$ in such a way that the band is contained within the block-tridiagonal structure, that we compute as dense.

The problem size n is defined by the number of block-rows, ℓ, and by the size of the blocks, k. The strong and weak scaling analyses have been obtained with a batch of experiments on which the number of processes vary in powers of two from 1 to 128, and ten different block sizes that vary from 64 up to 1024 have been tested. Table 3 details the block sizes used and the number of block-rows with each block size. When computing the weak scaling, as the problem size per process is maintained fixed, the number of block-rows per process depends exclusively on the block size (for a given matrix size). For the strong scaling, the number of block-rows depends on the number of processes used, on the block size and on the process index, as the total matrix size does not change.

We are interested in computing eigenvalues closest to the albedo parame-

ter. As we know that all the eigenvalues are smaller than the albedo, we use it as shift with the shift-and-invert technique operating on matrix $(A_n - \varpi I)^{-1}$, where $\varpi = 0.75$ in our runs. All tests compute 5 eigenvalues, except the runs corresponding to Fig. 8, which compute just 1 eigenvalue. In all cases, the relative residual norm of the computed eigenpairs, $\|Ax - \theta x\|/\|\theta x\|$, is always below the requested tolerance, `tol` $= 10^{-8}$. We have set the eigensolver to restart with a basis size of 16. In most runs, all eigenvalues converge without needing to restart, and hence 16 linear solves are performed per each factorization.

A final comment is that matrix A_n is not diagonally dominant in general. For the tests of Fig. 8, we have forced A_n to be artificially diagonally dominant, so that the reduced Spike method can be employed.

5.3. Performance results

Three sets of experiments show the weak scaling of the software versions. In the following figures we show the factorization time and the aggregated time of the multiple solves of the Arnoldi algorithm needed to obtain five eigenvalues working in double precision arithmetic for the smallest and the largest block sizes used.

The first set shows the time needed with the non-batched versions of the BCYCLIC algorithm for both approaches: explicitly computing the inverse of the diagonal block (_getri) or not (_getrs). The initial highlight is the better performance of the CPU executions with small block sizes and the better on the GPU with large block sizes. With small block sizes, the kernels executed on the GPU do not allow the devices to obtain their maximum performance.

In Figure 1 it is clear how explicitly computing the inverse takes more time during the factorization stage, and less in the solving stage, as could be expected. For larger block sizes the differences between the _getri and _getrs versions disappear in both stages, and the _getri times turn to be slightly smaller in the factorization while maintain the better performance during the solve, as can be seen in Figure 2.

The factorization with _getri on the CPU requires a block size larger than 512 to be faster than _getrs, while the GPU executions start to be faster with a block size larger than 128.

The second set of experiments shows exclusively executions on the GPU. We carry out a comparison of the five implementations that compute the inverse: the ones already used in the first set (_getri) and two more batched

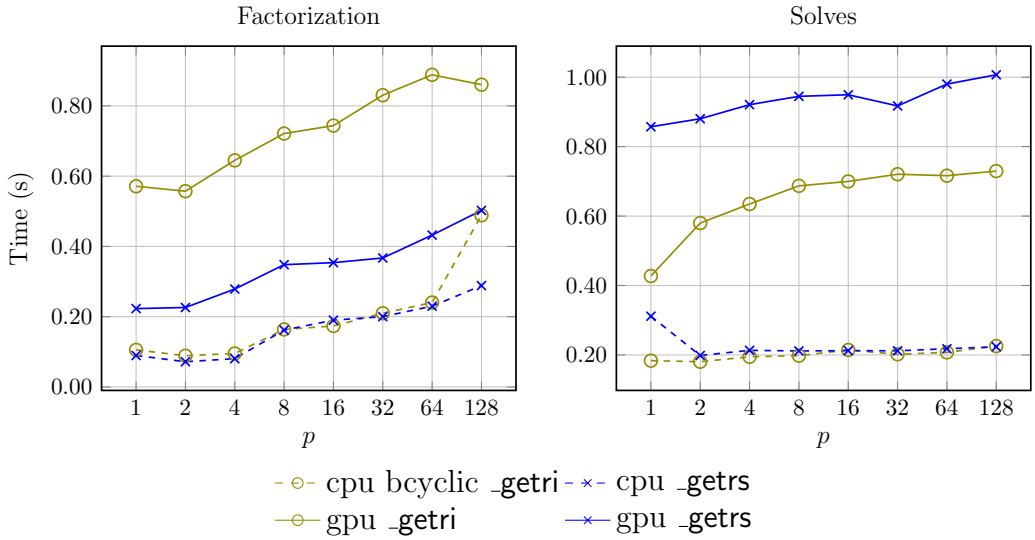

Figure 1: Weak scaling for the BCYCLIC algorithm running on CPU and on GPU with the non-batched version with $k = 64$ and $l = p \cdot 400$, where p is the number of MPI processes.

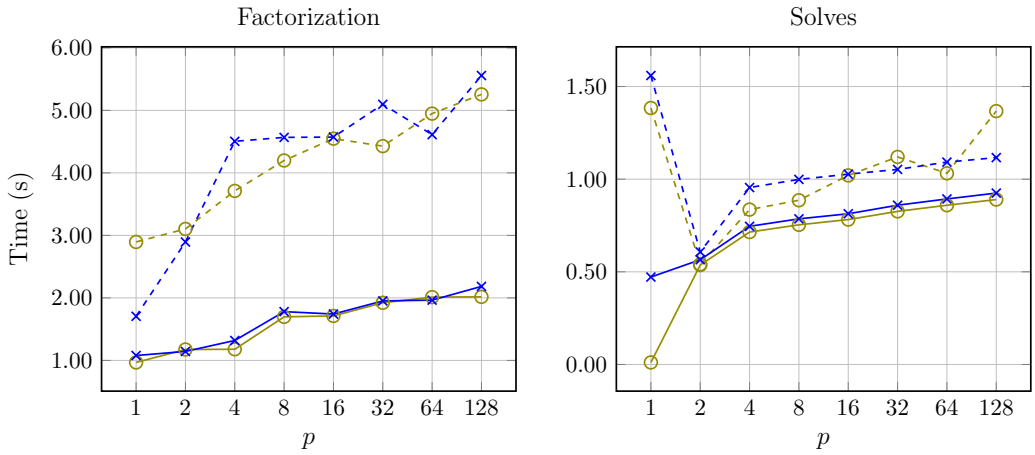

Figure 2: Weak scaling for the BCYCLIC algorithm running on CPU and on GPU with the non-batched version with $k = 1024$ and $l = p \cdot 25$, where p is the number of MPI processes. Consult the legend in Figure 1.

implementations (_getri_1 and _getri_2) per library used. Both batched versions share the same solve stage, so no different results should be seen in it.

Figure 3 allows us to see how for a small block size any of the batched versions performs faster than the non-batched one during the factorization. As expected, the batched versions allow the software to gain performance by computing the blocks in parallel. The solve results show a significant difference between the cuBLAS and MAGMA libraries due to their inherent performance, being cuBLAS faster in this stage.

The executions with a large block size seen in Figure 4 do not show the difference in time between the batched and non-batched implementations that occurs in the factorization with small sizes, as in this case, the large block size is enough to fulfill the massive parallel processing of the GPU. This occurs with block sizes larger than 512.

It is noticeable in Figure 4 how the cuBLAS implementations need considerably more time to perform the factorization while they are the fastest during the solve stage. These findings made us build a combined version of the software to benefit from the best of each libraries. This hybrid implementation selects the routines to use in the factorization based on the block size of the matrix. More precisely, cuBLAS _getri_2 is used with block sizes up to 128 and MAGMA _getri_1 for larger block sizes. The solve stage is managed by cuBLAS regardless of the block size.

Finally, the third set compares the performance of the BCYCLIC and the Spike algorithms running on the GPU and on the CPU. For the GPU versions, both algorithms use the combined implementation originated from the results of the previous set of experiments. For the sake of simplicity, an exception to the 'select the fastest functions' rule has been made, since for the case of using a block size smaller than 128 and no more than 4 processes the _getrs variant was slightly faster. Note that some executions of the Spike implementation with large block sizes could not run on the GPU due to memory constraints.

Figure 5 shows the behaviour of the algorithms with a small block size. Spike scales better than BCYCLIC in the factorization stage for this size and for sizes no larger than 128. That is evident in both the GPU and CPU executions. It starts being slower than BCYCLIC, but as soon as the number of processes is increased, it is able to obtain smaller times, needing a larger number of processes when executed on the GPU.

On the other hand, BCYCLIC scales worse due to the relatively higher

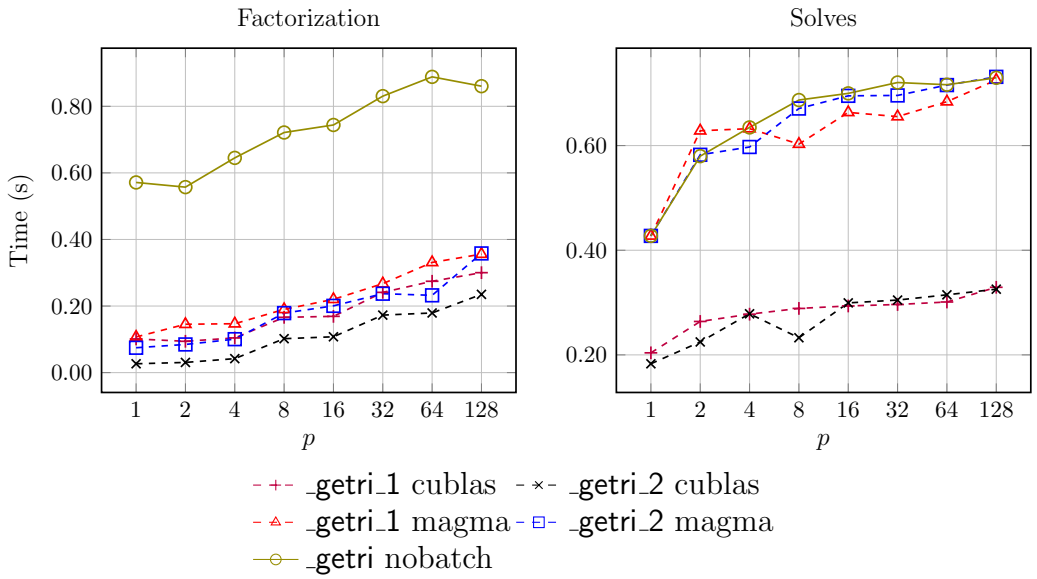

Figure 3: Weak scaling for the BCYCLIC algorithm running on GPU with batched and non-batched versions with $k = 64$ and $l = p \cdot 400$, where p is the number of MPI processes.

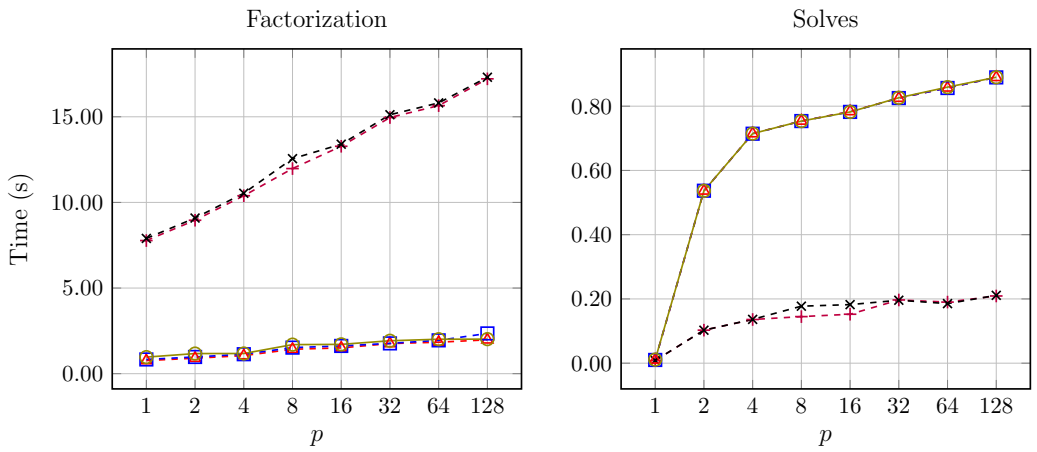

Figure 4: Weak scaling for the BCYCLIC algorithm running on GPU with batched and non-batched versions with $k = 1024$ and $l = p \cdot 25$, where p is the number of MPI processes. Consult the legend in Figure 3.

cost of communication with respect to the poor computing performance. We can see how the time increases noticeable when using several nodes (more than 4 processes). A block size larger than 128 is required for the BCYCLIC algorithm to perform better than Spike for any number of processes used during the factorization stage.

With all of the block sizes tested, the BCYCLIC algorithm has always performed faster than Spike in the solve stage. For small block sizes, the results highlight the better CPU performance with small BLAS-2 operations and the benefit of the absence of GPU-CPU data copies. From block sizes larger than 128, the payload of calling the kernels is worth the performance obtained with the GPU. On the CPU executions, the time gap between the two algorithms tends to increase when the block size grows while in the GPU executions it tends to decrease.

If the block size is increased up to 1024 as can be seen in Figure 6, the differences between the two algorithms and the two platforms are more prominent.

The different times obtained when varying the block size and the number of processes can be seen in more detail in Tables 4 and 5. The first one shows the total eigenproblem times obtained for all the different block sizes when using 128 processes. This table does not intend to contrast the performance obtained with different block sizes, as their computational cost differs and are not comparable in that sense. It allow us to compare the behaviour of the implementations for a specific block size. Spike is clearly faster with small block sizes as well as the executions on CPU. Once the block size exceeds 128, the executions on GPU with BCYCLIC obtain the smallest times.

Table 5 shows the total eigenproblem times for all the different number of processes when using the largest block size. When increasing the problem size by a factor of 128, the time increasing factor for the fastest implementation (BCYCLIC on GPU) is 3.3, whereas the same algorithm scales slightly better on CPU with a factor of 2.0. Even not being very close to a perfect scaling, these factors are reasonably good and contrast with the Spike factors, that double them.

The total eigenproblem time is also represented in Figure 7, in which we can highlight the case of using a block size of 640, where on CPU, the Spike algorithm performed faster than BCYCLIC against the normal behaviour, due to a drop in the performance of BCYCLIC with block sizes between 512 and 1024. From 640 and up to 1024 the BCYCLIC algorithm progressively recovers performance and obtains smaller times with larger block sizes.

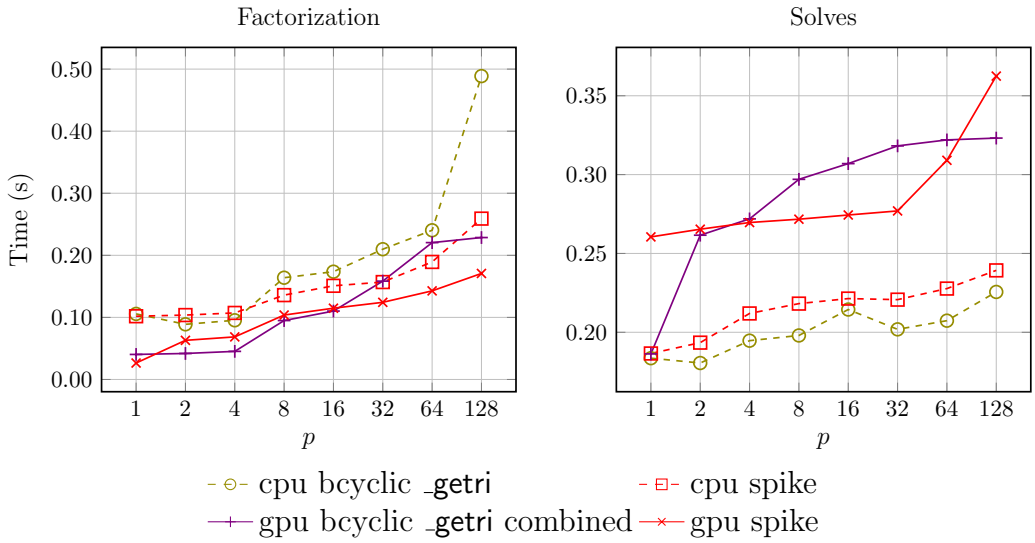

Figure 5: Weak scaling for the BCYCLIC and the Spike algorithms running on CPU and on GPU with $k = 64$ and $l = p \cdot 400$, where p is the number of MPI processes.

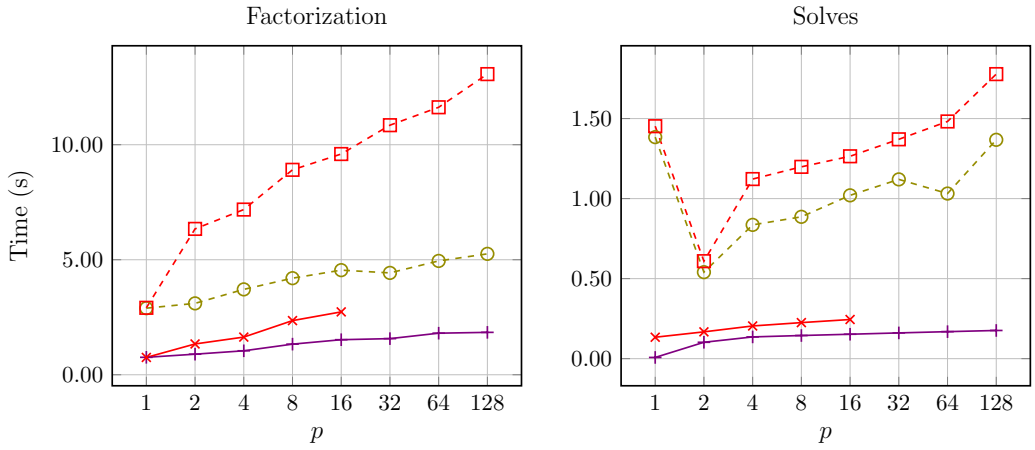

Figure 6: Weak scaling for the BCYCLIC and the Spike algorithms running on CPU and on GPU with $k = 1024$ and $l = p \cdot 25$, where p is the number of MPI processes. The executions on GPU with the Spike algorithm with more that 16 processes could not be done due to memory constraints. Consult the legend in Figure 5.

Table 4: Total eigenproblem time obtained with the weak scaling tests for 128 processes.

	CPU		GPU	
	Spike	BCYCLIC	Spike	BCYCLIC
Block size	Seconds	Seconds	Seconds	Seconds
64	**0.67**	1.18	0.68	0.85
96	0.96	1.42	**0.75**	0.81
128	0.85	1.04	**0.75**	0.76
256	1.34	1.88		**1.13**
384	3.04	2.50		**1.06**
512	5.12	3.91		**1.90**
640	7.83	10.46		**6.91**
768	14.05	9.62		**6.03**
896	17.06	8.76		**3.55**
1024	22.08	8.42		**3.01**

Table 5: Total eigenproblem time obtained with the weak scaling tests for a block size $k = 1024$.

	CPU		GPU	
	Spike	BCYCLIC	Spike	BCYCLIC
Processes	Seconds	Seconds	Seconds	Seconds
1	4.37	4.29	**0.91**	0.91
2	8.07	3.70	1.68	**1.12**
4	10.22	4.78	2.32	**1.38**
8	13.17	5.57	3.76	**1.85**
16	14.91	6.36	4.55	**2.21**
32	17.22	6.50		**2.39**
64	19.12	7.37		**2.89**
128	22.08	8.42		**3.01**

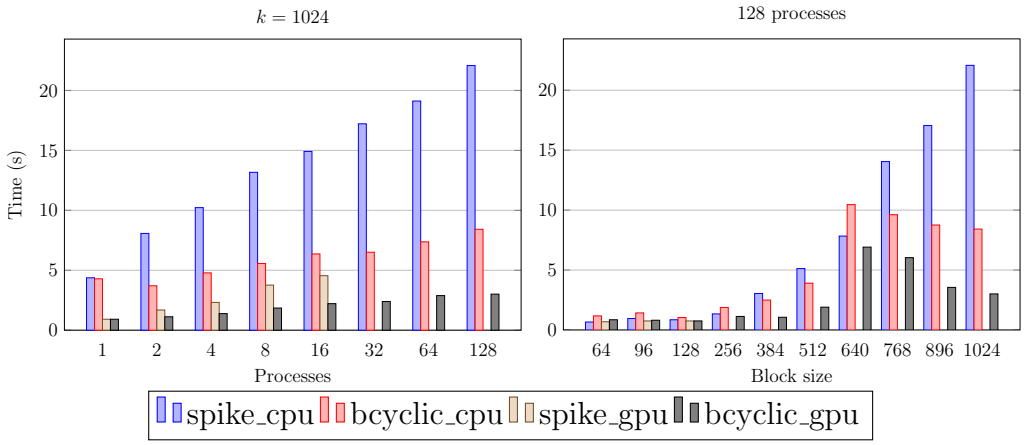

Figure 7: Total eigenproblem time obtained with the weak scaling tests for a block size $k = 1024$ (left) and for 128 processes (right).

The experiments to measure the strong scaling use diagonally dominant matrices and compare the performance of the already tested implementations with the reduced Spike, used as a direct solver. The figures represent the results obtained when measuring the strong scaling for a fixed matrix size of 307200[4]. For the case of one process, this size is larger than the one used in weak scaling tests, in order to have enough workload with a reasonable number of processes.

Since for the strong scaling tests the time needed to complete the solve stage does not vary significantly between the algorithms, we present the total eigenvalue problem solve operation time in the figures, that have an almost direct correspondence with the time needed to perform the factorization stage and at the same time provide us with a more global view.

Figure 8 shows the strong scaling results for three different block sizes. Again, for small block sizes where the GPU has a large overhead launching a lot of small kernels, the CPU times tend to be smaller. And when the block size is increased, the same algorithm performs faster on GPU. For a block size of 64, the three algorithms scale well up to 8 processes, and from that point on the performance of BCYCLIC decays due to a higher ratio between communication and computation time. Spike achieves a better scalability

[4]The actual size of the matrix when using a block size of 896 is 307328.

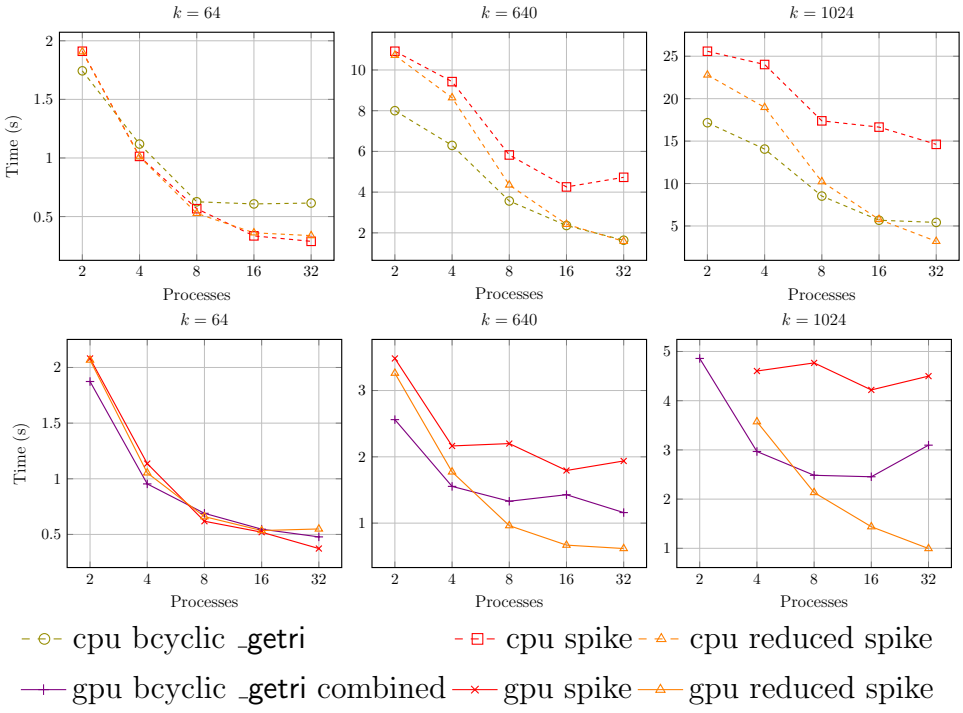

Figure 8: Strong scaling for the BCYCLIC, the Spike and the reduced Spike algorithms running on CPU and on GPU with a total matrix dimension of 307200 and different block sizes k.

than the other two algorithms.

The middle and right plots in Figure 8 show that the scalability of the algorithms with larger block sizes is not good. Spike turns into the slowest of the algorithms and the one with the worst scalability. On the other side, the reduced Spike benefits of a larger block size scaling up to a larger number of processes where the BCYCLIC algorithm performance starts to decay.

6. Conclusions and future work

We have developed a set of codes for computing a few eigenpairs of large-scale matrices with tridiagonal structure via Krylov methods. Most of the developing effort has been concentrated on the scalable solution of block-tridiagonal linear systems. The codes are integrated in the SLEPc/PETSc

framework, with an MPI-CUDA programming style, and allow to use many processors/GPUs to address very large-scale problems.

Performance analysis allows us to draw several conclusions. In general, BCYCLIC performs better than Spike, but Spike scales better when using small block sizes. All GPU implementations have shown to be faster than the CPU counterparts, except for small block sizes. In terms of scalability, we can state that for sufficiently large block sizes, the codes scale well for up to 128 MPI processes (GPUs). Our implementations can use either cuBLAS or MAGMA, or a combination of the two. The best performance has been obtained with the mixed implementation.

Another conclusion is that, for large block size, BCYCLIC is able to solve larger problem sizes with respect to Spike, because Spike has larger memory requirements. Finally, for the case of diagonally dominant block-tridiagonal matrices, the reduced Spike method achieves better scalability.

Although the performance analysis was carried out in the context of Krylov eigensolvers, the conclusions could be applied to other applications where a sequence of linear systems with block-tridiagonal matrices must be solved, since in our tests almost all computation is associated with the factorization and linear solves. The solvers can also be used with banded matrices, in which case the off-diagonal blocks are triangular (although we have not exploited this fact).

As a work in progress, we mention that we are extending our codes for the case that periodic boundary conditions make the matrix block-tridiagonal with an additional block in the upper-right and lower-left corners. Future work also includes a further optimization of the orthogonalization phase in Arnoldi, possibly with the implementation of fused kernels as in [38].

Acknowledgements. The authors would like to thank the reviewers for their careful reading and helpful suggestions. The simulations corresponding to section 5 were carried out on the supercomputer Minotauro, belonging to the Spanish Supercomputing Network (RES). The test matrices were kindly provided by Paulo Vasconcelos.

References

[1] S. P. Hirshman, K. S. Perumalla, V. E. Lynch, R. Sanchez, BCYCLIC: A parallel block tridiagonal matrix cyclic solver, J. Comput. Phys. 229 (18) (2010) 6392–6404.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, H. van der Vorst (Eds.), Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide, Society for Industrial and Applied Mathematics, Philadelphia, PA, 2000.

[3] P. Bientinesi, F. D. Igual, D. Kressner, M. Petschow, E. S. Quintana-Ortí, Condensed forms for the symmetric eigenvalue problem on multithreaded architectures, Concur. Comput.: Pract. Exp. 23 (2011) 694–707.

[4] S. Tomov, R. Nath, J. Dongarra, Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing, Parallel Comput. 36 (12) (2010) 645–654.

[5] C. Vomel, S. Tomov, J. Dongarra, Divide and conquer on hybrid GPU-accelerated multicore systems, SIAM J. Sci. Comput. 34 (2) (2012) C70–C82.

[6] A. Haidar, H. Ltaief, J. Dongarra, Toward a high performance tile divide and conquer algorithm for the dense symmetric eigenvalue problem, SIAM J. Sci. Comput. 34 (6) (2012) C249–C274.

[7] H. Anzt, S. Tomov, J. Dongarra, On the performance and energy efficiency of sparse linear algebra on GPUs, Int. J. High Perform. Comput. Appl. To appear.
URL `https://doi.org/10.1177/1094342016672081`

[8] J. I. Aliaga, P. Alonso, J. M. Badía, P. Chacón, D. Davidović, J. R. López-Blanco, E. S. Quintana-Ortí, A fast band-Krylov eigensolver for macromolecular functional motion simulation on multicore architectures and graphics processors, J. Comput. Phys. 309 (2016) 314–323.

[9] V. Hernandez, J. E. Roman, V. Vidal, SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems, ACM Trans. Math. Software 31 (3) (2005) 351–362.

[10] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. Gropp, D. Karpeyev, D. Kaushik, M. Knepley, L. C. McInnes, K. Rupp, B. Smith, S. Zampini, H. Zhang, H. Zhang, PETSc users manual, Tech. Rep. ANL-95/11 - Revision 3.7, Argonne National Laboratory (2016).

[11] V. Minden, B. Smith, M. G. Knepley, Preliminary implementation of PETSc using GPUs, in: D. A. Yuen, et al. (Eds.), GPU solutions to multi-scale problems in science and engineering, Springer, 2013, pp. 131–140.

[12] S. Dalton, N. Bell, L. Olson, M. Garland, Cusp: Generic parallel algorithms for sparse matrix and graph computations, version 0.5.0 (2014). URL `http://cusplibrary.github.io/`

[13] NVIDIA, CUBLAS Library V7.0, Tech. Rep. DU-06702-001_v7.0, NVIDIA Corporation (2015).

[14] A. Lamas Daviña, E. Ramos, J. E. Roman, Optimized analysis of isotropic high-nuclearity spin clusters with GPU acceleration, Comput. Phys. Commun. 209 (2016) 70–78.

[15] A. Lamas Daviña, J. E. Roman, GPU implementation of Krylov solvers for block-tridiagonal eigenvalue problems, in: R. Wyrzykowski, et al. (Eds.), Parallel Processing and Applied Mathematics–PPAM 2015, Part I, Vol. 9573 of Lect. Notes Comp. Sci., Springer, 2016, pp. 182–191.

[16] G. W. Stewart, A Krylov–Schur algorithm for large eigenproblems, SIAM J. Matrix Anal. Appl. 23 (3) (2001) 601–614.

[17] E. Gallopoulos, B. Philippe, A. H. Sameh, Parallelism in Matrix Computations, Springer, Dordrecht, 2016.

[18] B. L. Buzbee, G. H. Golub, C. W. Nielson, On direct methods for solving Poisson's equations, SIAM J. Numer. Anal. 7 (4) (1970) 627–656.

[19] J. J. Lambiotte, Jr., R. G. Voigt, The solution of tridiagonal linear systems on the CDC STAR 100 computer, ACM Trans. Math. Software 1 (4) (1975) 308–329.

[20] W. Gander, G. H. Golub, Cyclic reduction: history and applications, in: R. J. P. Franklin T. Luk (Ed.), Proceedings of the Workshop on Scientific Computing, Springer, 1997, pp. 73–85.

[21] D. Heller, Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems, SIAM J. Numer. Anal. 13 (4) (1976) 484–496.

[22] P. Yalamov, V. Pavlov, Stability of the block cyclic reduction, Linear Algebra Appl. 249 (1) (1996) 341–358.

[23] E. Polizzi, A. H. Sameh, A parallel hybrid banded system solver: the SPIKE algorithm, Parallel Comput. 32 (2) (2006) 177–194.

[24] C. C. K. Mikkelsen, M. Manguoglu, Analysis of the truncated SPIKE algorithm, SIAM J. Matrix Anal. Appl. 30 (4) (2009) 1500–1519.

[25] K. Mendiratta, E. Polizzi, A threaded SPIKE algorithm for solving general banded systems, Parallel Comput. 37 (12) (2011) 733–741.

[26] Y. Zhang, J. Cohen, J. D. Owens, Fast tridiagonal solvers on the GPU, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '10, 2010, pp. 127–136.

[27] S. K. Seal, K. S. Perumalla, S. P. Hirshman, Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations, J. Parallel and Distrib. Comput. 73 (2) (2013) 273–280.

[28] A. J. Park, K. S. Perumalla, Efficient heterogeneous execution on large multicore and accelerator platforms: Case study using a block tridiagonal solver, J. Parallel and Distrib. Comput. 73 (12) (2013) 1578–1591.

[29] S. Tomov, J. Dongarra, M. Baboulin, Towards dense linear algebra for hybrid GPU accelerated manycore systems, Parallel Comput. 36 (5-6) (2010) 232–240.

[30] B. Baghapour, V. Esfahanian, M. Torabzadeh, H. M. Darian, A discontinuous Galerkin method with block cyclic reduction solver for simulating compressible flows on GPUs, Int. J. Comput. Math. 92 (1) (2015) 110–131.

[31] E. László, M. Giles, J. Appleyard, Manycore algorithms for batch scalar and block tridiagonal solvers, ACM Trans. Math. Software 42 (4) (2016) 31:1–31:36.

[32] L.-W. Chang, J. A. Stratton, H.-S. Kim, W.-M. W. Hwu, A scalable, numerically stable, high-performance tridiagonal solver using GPUs, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, 2012, pp. 27:1–27:11.

[33] I. Venetis, A. Kouris, A. Sobczyk, E. Gallopoulos, A. Sameh, A direct tridiagonal solver based on Givens rotations for GPU architectures, Parallel Comput. 49 (2015) 101–116.

[34] I. E. Venetis, A. Sobczyk, A. Kouris, A. Nakos, N. Nikoloutsakos, E. Gallopoulos, A general tridiagonal solver for coprocessors: Adapting g-Spike for the Intel Xeon Phi, in: G. R. Joubert, et al. (Eds.), Parallel Computing: On the Road to Exascale, IOS Press, 2015, pp. 371–380.

[35] R. Serban, D. Melanz, A. Li, I. Stanciulescu, P. Jayakumar, D. Negrut, A GPU-based preconditioned Newton–Krylov solver for flexible multibody dynamics, Internat. J. Numer. Methods Engrg. 102 (9) (2015) 1585–1604.

[36] M. Ahues, F. D. d'Almeida, A. Largillier, O. Titaud, P. Vasconcelos, An L^1 refined projection approximate solution of the radiation transfer equation in stellar atmospheres, J. Comput. Appl. Math. 140 (2002) 13–26.

[37] P. B. Vasconcelos, O. Marques, J. E. Roman, Parallel eigensolvers for a discretized radiative transfer problem, in: J. M. L. M. Palma, et al. (Eds.), High Performance Computing for Computational Science–VECPAR 2008, Vol. 5336 of Lect. Notes Comp. Sci., Springer, 2008, pp. 336–348.

[38] K. Rupp, J. Weinbub, A. Jüngel, T. Grasser, Pipelined iterative solvers with kernel fusion for graphics processing units, ACM Trans. Math. Software 43 (2) (2016) 11:1–11:27.