Document downloaded from:

http://hdl.handle.net/10251/66516

This paper must be cited as:

Lamas Daviña, A.; Román Moltó, JE. (2016). GPU implementation of Krylov solvers for block-tridiagonal eigenvalue problems. En Parallel Processing and Applied Mathematics. Springer. 182-191. doi:10.1007%2F978-3-319-32149-3_18

The final publication is available at

http://link.springer.com/chapter/10.1007%2F978-3-319-32149-3_18

Copyright Springer

Additional Information

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-32149-3_18

GPU implementation of Krylov solvers for block-tridiagonal eigenvalue problems*

Alejandro Lamas Daviña and Jose E. Roman

Universitat Politècnica de València
D. Sistemes Informàtics i Computació, Camí de Vera s/n, 46022 València, Spain
{alejandro.lamas,jroman}@dsic.upv.es

Abstract. In an eigenvalue problem defined by one or two matrices with block-tridiagonal structure, if only a few eigenpairs are required it is interesting to consider iterative methods based on Krylov subspaces, even if matrix blocks are dense. In this context, using the GPU for the associated dense linear algebra may provide high performance. We analyze this in an implementation done in the context of SLEPc, the Scalable Library for Eigenvalue Problem Computations. In the case of a generalized eigenproblem or when interior eigenvalues are computed with shift-and-invert, the main computational kernel is the solution of linear systems with a block-tridiagonal matrix. We explore possible implementations of this operation on the GPU, including a block cyclic reduction algorithm.

Keywords: GPU computing, eigenvalue computation, Krylov methods, block-tridiagonal linear solvers

1 Introduction

This paper is concerned with the computation of a few eigenpairs of an eigenvalue problem defined by one or two matrices with block-tridiagonal structure, using graphic processing units (GPU). We focus on Krylov methods, that will be competitive with respect to other methods when the percentage of wanted eigenvalues is small. We remark that our solvers are not restricted to symmetric matrices, but can work also in the non-symmetric case.

Given an $n \times n$ real matrix A, the standard eigenvalue problem is formulated as

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

where λ is a scalar (eigenvalue) and $x \neq 0$ is an n-vector (eigenvector). There are n eigenpairs (x, λ) satisfying (1). If matrix A is symmetric, then all eigenvalues are real, otherwise eigenvalues are complex in general. In the generalized eigenvalue problem there are two intervening matrices,

$$Ax = \lambda Bx. \tag{2}$$

* This work was partially supported by the Spanish Ministry of Economy and Competitiveness under grant TIN2013-41049-P. Alejandro Lamas was supported by the Spanish Ministry of Education, Culture and Sport through grant FPU13-06655.

There are two major strategies for solving the above eigenproblems. One class of methods first reduce the matrices to a condensed form from which eigenvalues can be recovered more easily. These methods are generally more appropriate when all eigenvalues are required. In contrast, a second class of methods based on projecting the eigenproblem on a low-dimensional subspace are usually better suited for computing only a few eigenpairs. For large, sparse matrices, projection methods are the only viable strategy because they preserve sparsity, as opposed to transformation methods that produce fill-in during reduction to condensed form. In this work, we are targeting problems where matrices have a block-tridiagonal structure, whose blocks are not necessarily sparse, in which case transformation methods are in principle well suited, but we take the projection route since we are interested in just a few eigenvalues.

Consider first the standard eigenproblem (1). Transformation methods begin by reducing matrix A to either tridiagonal or upper Hessenberg form, in the symmetric or non-symmetric case, respectively. In terms of computational effort, this step is more expensive than the actual computation of eigenvalues, and hence many research efforts have been dedicated to improve its arithmetic intensity, specifically on GPUs [2],[14]. Once the problem has been reduced to condensed form, various algorithms can be applied to compute the eigenvalues, such as the QR iteration, or specific iterations for symmetric tridiagonals, such as divide-and-conquer. Some of these algorithms are difficult to implement and have modest arithmetic intensity, and hence a hybrid CPU-GPU approach is often pursued [15]. In order to increase arithmetic intensity, some authors extend the algorithms to operate directly on a symmetric band matrix [3]. These methods compute all eigenvalues, which may be wasteful in some applications, and, optionally, all eigenvectors (requiring an additional computation).

One example of the projection methods is the Arnoldi algorithm: starting with v_1, $\|v_1\|_2 = 1$, the Arnoldi basis generation process can be expressed by the recurrence

$$v_{j+1}h_{j+1,j} = w_j = Av_j - \sum_{i=1}^{j} h_{i,j}v_i, \tag{3}$$

where $h_{i,j}$ are the scalar coefficients obtained in the Gram-Schmidt orthogonalization of Av_j with respect to v_i, $i = 1, 2, \ldots, j$, and $h_{j+1,j} = \|w_j\|_2$. The columns of V_j span the Krylov subspace $\mathcal{K}_j(A, v_1) = \mathrm{span}\{v_1, Av_1, A^2v_1, \ldots, A^{j-1}v_1\}$ and $Ax = \lambda x$ is projected into $H_j s = \theta s$, where H_j is an upper Hessenberg matrix with elements $h_{i,j}$ ($h_{i,j} = 0$ for $i \geq j+2$). If the solutions of the projected eigenproblem are assumed to be (θ_i, s_i), $i = 1, 2, \ldots, j$, then the approximate eigenpairs $(\tilde{x}_i, \tilde{\lambda}_i)$ of the original problem are obtained as $\tilde{\lambda}_i = \theta_i,\ \tilde{x}_i = V_j s_i$.

Regarding the implementation of projection methods for eigenvalue problems on the GPU, particularly Krylov methods, the challenge is to reach high Gflops rate in the sparse matrix-vector product operation, see e.g. [10], since the rest of operations are quite simple as will be discussed in Section 3. In our case, we will not deal with sparse matrices, and the goal will be to implement highly efficient computational kernels for the block-tridiagonal matrix-vector product.

When addressing the generalized eigenproblem (2), a possible approach is to transform it to the standard form and apply the methods discussed above, for instance solve $B^{-1}Ax = \lambda x$ (provided B is non-singular). In the context of projection methods, where only a few eigenvalues are computed, the shift-and-invert transformation is commonly used,

$$(A - \sigma B)^{-1} Bx = \theta x, \tag{4}$$

where largest magnitude $\theta = (\lambda - \sigma)^{-1}$ correspond to eigenvalues λ closest to a given target value σ. Rather than computing matrix $(A - \sigma B)^{-1}B$ explicitly, Krylov methods normally operate implicitly by solving linear systems with $A - \sigma B$ when necessary. In our case, we need an efficient kernel to solve linear systems with a block-tridiagonal coefficient matrix on the GPU.

The rest of the paper is organized as follows. In section 2 we describe the SLEPc library, in which we have developed our solvers, focusing on the support for GPU computing. Section 3 provides details of our implementation, paying special attention to the kernels for matrix-vector products and linear system solves with block-tridiagonals. Results of some computational experiments are shown in section 4. Finally, we close with some concluding remarks.

2 SLEPc solvers on GPU

SLEPc, the Scalable Library for Eigenvalue Problem Computations [5], is a software package for the solution of large-scale eigenvalue problems on parallel computers. It can be used to solve standard and generalized eigenproblems, (1) and (2), as well as other related problems. It can work with either real or complex arithmetic, in single or double precision. SLEPc provides a collection of eigensolvers, most of which are based on the subspace projection paradigm described in the previous section. In particular, it includes a robust and efficient parallel implementation of a restarted Krylov solver. It also supports the shift-and-invert transformation (4) with which interior eigenvalues can be computed making use of the linear solvers provided by PETSc[1].

In the development version, PETSc incorporates support for NVIDIA GPUs by means of Thrust and CUSP[2], performing vector operations and matrix-vector products through VECCUSP, a special vector class whose array is mirrored in the GPU, and a matrix class MATAIJCUSP, where data generated on the host is then copied to the device on demand. Later, support was extended for sparse matrix operations via CUSPARSE. The GPU model considered in PETSc uses MPI for communication between different processes, each of them having access to a single GPU [7]. The implementation includes mechanisms to guarantee coherence of the mirrored data-structures in the host and the device.

---

[1] http://www.mcs.anl.gov/petsc

[2] Thrust is a C++ template library included in the CUDA software development toolkit that makes common CUDA operations concise and readable. CUSP is an open source library based on Thrust that targets sparse linear algebra.

In a previous work [11], preliminary support for GPU computing on SLEPc was analyzed in the context of an application arising from an integral equation. In this work, we extend the developments to general block-tridiagonal matrices including shift-and-invert.

3 Krylov methods for block-tridiagonal matrices

Krylov algorithms for eigenvalue computations are based on building an orthogonal basis of the Krylov subspace $\mathcal{K}_m(A, v_1)$ and then performing a Rayleigh-Ritz projection to extract approximate eigenpairs. Since convergence may be slow, it is necessary to restart the method, that is, discard part of the information contained in the subspace and extend the subspace again. We use the Krylov-Schur restart [13]. We will not describe the algorithm in detail here, just enumerate the main computational kernels:

1. Basis expansion. To obtain a new candidate vector for the Krylov subspace, a previous vector must be multiplied by A. In the generalized eigenproblem (2) the multiplication is by $B^{-1}A$ or, alternatively, by $(A - \sigma B)^{-1}B$ and hence linear system solves are required as discussed previously.
2. Orthogonalization and normalization of vectors. The jth vector of the Krylov basis must be orthogonalized against the previous $j - 1$ vectors. This can be done with the (iterated) modified or classical Gram-Schmidt procedure.
3. Solution of projected eigenproblem. A small eigenvalue problem of size m must be solved at each restart, for matrix $H = V^T AV$ (already available from previous steps).
4. Restart. The associated computation is VZ, where the columns of V span the Krylov subspace and Z is a small matrix of order $m \times r$, with $r < m$, formed by the Schur vectors of H (calculated on the previous step).

We assume that m is very small compared to the size of the matrices, n, and we can have for instance $m = 30$, $n = 100000$. The cost of item 3 in the above list is negligible compared to the rest, and hence it is not worth performing it on the GPU. Regarding items 2 and 4, they are currently done as vector operations (BLAS1) on the GPU. We plan to optimize these operations in the future so that they use BLAS level 2 (or even level 3 in some cases), but this is not very relevant in this paper because the dominant cost is by far the one associated with item 1. Next we discuss in detail the operations related to basis expansion.

3.1 Kernel: matrix-vector product

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

$$T = \begin{bmatrix} B_1 & C_1 & & & \\ A_2 & B_2 & C_2 & & \\ & A_3 & B_3 & C_3 & \\ & & \ddots & \ddots & \ddots \\ & & & A_\ell & B_\ell \end{bmatrix}, \qquad \mathrm{rep}(T) = \begin{bmatrix} \square & B_1 & C_1 \\ A_2 & B_2 & C_2 \\ A_3 & B_3 & C_3 \\ \vdots & \vdots & \vdots \\ A_\ell & B_\ell & \square \end{bmatrix}, \tag{5}$$

where rep(T) stands for the memory representation of T (the $\square$ symbols indicate blocks with memory allocated but not being used).

The matrix-vector product $y = Tv$ can be computed by blocks as

$$y_i = [A_i \; B_i \; C_i] \begin{bmatrix} v_{i-1} \\ v_i \\ v_{i+1} \end{bmatrix}, \qquad i = 2, \dots, \ell - 1, \tag{6}$$

with analog expressions for the first and last block row. Due to the arrangement of blocks in rep(T), it is possible to do the computation with a single call to BLAS _gemv per each block row. Similarly, a possible GPU implementation would call the corresponding CUBLAS [8] subroutine. We remark that when allocating memory for rep(T) on the GPU we use appropriate 2D padding for each block, to guarantee alignment of columns. Apart from the CUBLAS version, we have also implemented a customized kernel that performs the whole computation with a single kernel invocation.

3.2 Kernel: linear system solves

We now turn our attention to the solution of linear systems of equations

$$Tx = b. \tag{7}$$

This problem could be approached with LAPACK's general band factorization subroutines, but this is not available on the GPU. Hence, we focus on algorithms that operate specifically on the block-tridiagonal structure and are feasible to implement with CUDA. The (scalar) tridiagonal case was analyzed in [16], where the authors compare GPU implementations of several algorithms. We have extended two of the algorithms to the block case: Thomas and cyclic reduction.

Gaussian elimination on a tridiagonal matrix is sometimes referred to as the Thomas algorithm. We have implemented the block version of this method just for reference, because it is an inherently serial algorithm, with little opportunity of parallelism except for computations within one block. In the forward elimination phase, the algorithm computes $C_1 \leftarrow B_1^{-1} C_1$ and $b_1 \leftarrow B_1^{-1} b_1$, and

$$B_i \leftarrow B_i - A_i C_{i-1}, \tag{8}$$

$$C_i \leftarrow B_i^{-1} C_i, \tag{9}$$

$$b_i \leftarrow B_i^{-1}(b_i - A_i b_{i-1}), \tag{10}$$

for $i = 2, \dots, \ell$; the backward substitution starts with $x_l \leftarrow b_l$ and runs for $i = \ell - 1, \dots, 1$

$$x_i \leftarrow b_i - C_i x_{i+1}. \tag{11}$$

Steps (8)–(9) perform a block LU factorization, that needs to be computed only once. Subsequent right-hand sides only require steps (10)–(11). Note that the factorization is destructive, but we assume that the original matrix T is no longer needed. Factorization can be accomplished with a few calls to _gemm

and LAPACK's _getrf/_getrs. With _getrf we compute the LU factorization with partial pivoting of the diagonal block B_i. We remark that since pivoting is limited to the diagonal block, this algorithm is numerically less robust than a full LU factorization, although in our tests the computed result was always accurate enough.

There are several alternative algorithms that try to increase the number of concurrent tasks and hence reduce the length of the critical path, although the cost in flops is increased. The cyclic reduction scheme [4] consists of $\log_2 \ell$ stages, where in every stage j all even blocks are eliminated in terms of the odd blocks, resulting in a system with a similar form but with halved number of unknowns. Matrix blocks of consecutive levels are related by

$$A_i^{(j+1)} = -A_{2i}^{(j)}(B_{2i-1}^{(j)})^{-1}A_{2i-1}^{(j)}, \tag{12}$$

$$B_i^{(j+1)} = B_{2i}^{(j)} - A_{2i}^{(j)}(B_{2i-1}^{(j)})^{-1}C_{2i-1}^{(j)} - C_{2i}^{(j)}(B_{2i+1}^{(j)})^{-1}A_{2i+1}^{(j)}, \tag{13}$$

$$C_i^{(j+1)} = -C_{2i}^{(j)}(B_{2i+1}^{(j)})^{-1}C_{2i+1}^{(j)}, \tag{14}$$

and analog recurrences for the right-hand side $b_i^{(j+1)}$ during forward elimination, and the solution vector $x_i^{(j+1)}$ during backward substitution. This algorithm also requires more memory, since $(B_{2i-1}^{(j)})^{-1}A_{2i-1}^{(j)}$ and $(B_{2i+1}^{(j)})^{-1}C_{2i+1}^{(j)}$ are necessary also during backward substitution. The other computed quantities can be stored in-place, so the storage requirements are at least 66% higher than Thomas.

The GPU implementation requires a version of CUBLAS that provides the _getrf and _getrs operations, and more precisely *batched* versions of them that allow launching several factorizations/triangular solves simultaneously in order to increase potential parallelism. In this sense, the Thomas algorithm is very poor, since only one factorization or triangular solve can be done at a time.

An MPI parallel implementation of the cyclic reduction for block-tridiagonals was considered in [6],[12]. A block cyclic reduction solver on GPU was used in [1] in the context of a CFD applications, with block sizes up to 32. We are interested in the case of much larger blocks, as was done in [9] in the context of a hybrid CPU-GPU solver based on MAGMA for the LU factorization.

4 Computational results

The tests have been run on two computers:

Fermi 2 Intel Xeon E5649 processor (6 cores) at 2.53 GHz, 24 GB of main memory; 2 GPUs NVIDIA Tesla M2090, 512 cores and 6 GB GDDR per GPU. The operating system is RHEL 6.0, with GCC 4.6.1 and MKL 11.1.

Kepler 2 Intel Core i7 3820 processor (2 cores) at 3,60 GHz with 16 GB of main memory; 2 GPUs NVIDIA Tesla K20c, with 2496 cores and 5 GB GDDR per GPU. The operating system is CentOS 6.6, with GCC 4.4.7 and MKL 11.0.2.

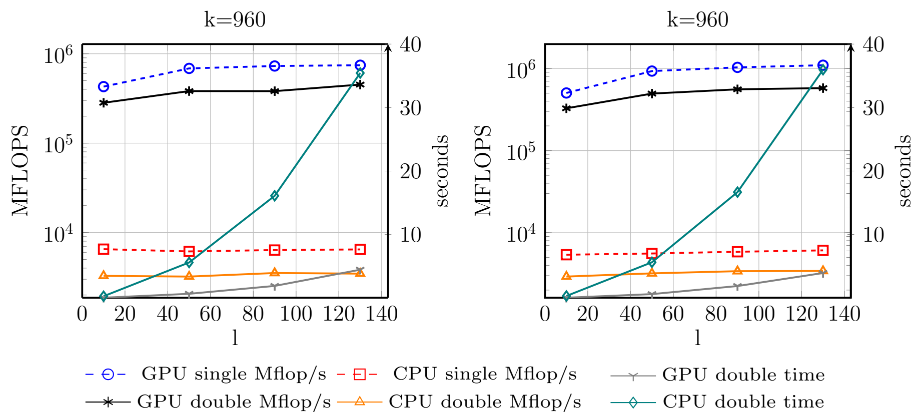

Fig. 1. Left y axis: Performance of the matrix-vector product operation for a fixed block size $k = 960$ and varying number of blocks ℓ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in single and double precision arithmetic. Right y axis: Eigensolve operation time for double precision arithmetic.

In both cases, the other software used is PETSc 3.6, SLEPc 3.6, CUDA 7.0 and CUSP 0.5.0.

Computational experiments have been conducted on random matrices, where we have varied the number of blocks ℓ and the block size k, up to the maximum storage space available on the GPU card. The matrices were generated in all cases on the CPU to use the exact same matrix on both runs (GPU/CPU).

We start discussing the matrix-vector product operation, implemented in the CPU with calls to BLAS and in the GPU with the ad-hoc CUDA kernel. We have computed the largest magnitude eigenvalue of random matrices, where the computation requires about 100 matrix-vector products. Figures 1 and 2 show the Mflop/s rate (left y axis) for our code running either on the CPU (with MKL and multi-thread enabled) or the GPU, and the total eigensolve operation time (right y axis). With a large block size (Figure 1), we can see that performance does not depend too much on the number of blocks. In contrast, when we fix the number of blocks (Figure 2) the performance is significantly lower for small block sizes. In any case, the benefit of using the GPU is evident since we are able to reach about 1 Tflop/s.

For assessing the performance of the shift-and-invert computation using the block oriented cyclic reduction algorithm, we have computed one eigenvalue closest to the origin ($\sigma = 0$) of random matrices of varying size. Results are shown in Figures 3 and 4. In this case, the GPU version does not beat the CPU computation (using as many threads as computational cores in MKL operations). Nevertheless we can appreciate the sensitiveness of the GPU to the data size, as its performance improves when the number of blocks is increased for a large block size as well as when the block size is increased, while the CPU is only affected

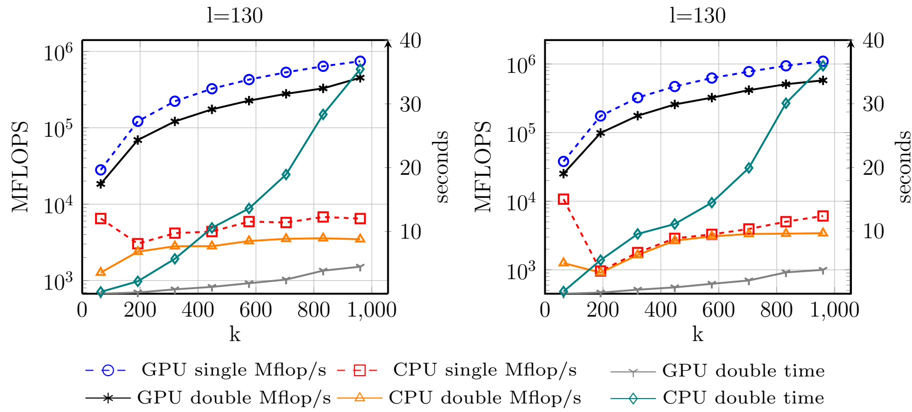

Fig. 2. Left y axis: Performance of the matrix-vector product operation for varying block size k and a fixed number of blocks $\ell = 130$ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in single and double precision arithmetic. Right y axis: Eigensolve operation time for double precision arithmetic.

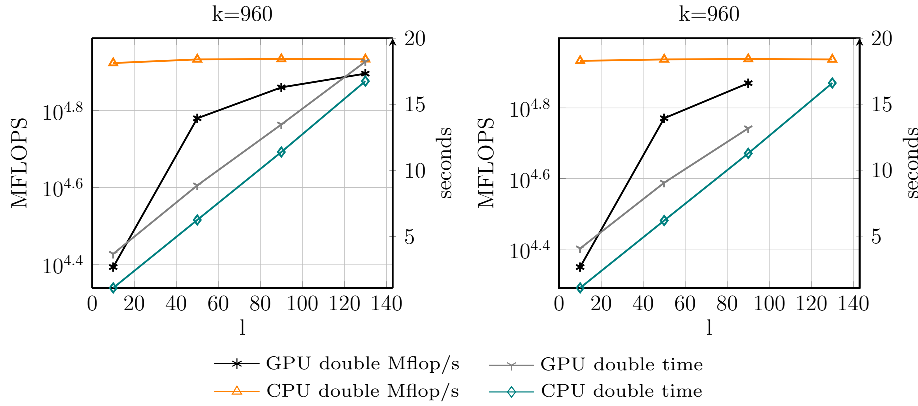

Fig. 3. Shift-and-invert case. Left y axis: Performance of factorization for a fixed block size k and varying number of blocks ℓ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic. Right y axis: Eigensolve operation time for double precision arithmetic.

by the block size. The reported Mflop/s rate correspond to the LU factorization on double precision arithmetic, whereas the triangular solves only achieve a performance around 2.5-4 Gflop/s, both on CPU and GPU, and the single precision tests provided inaccurate results. The time shown corresponds to the total eigensolve operation, using the same number of restarts on both GPU and

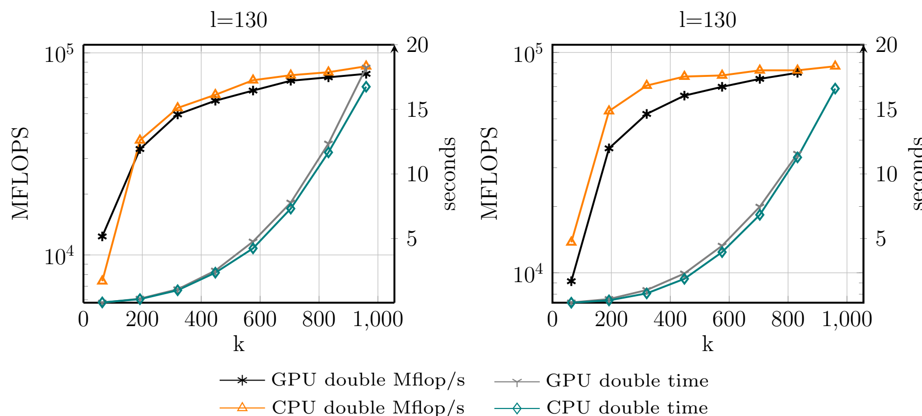

Fig. 4. Shift-and-invert case. Left y axis: Performance of factorization for varying block size k and a fixed number of blocks $\ell = 130$ for both CPU and GPU on the Fermi (left) and Kepler (right) machine, in double precision arithmetic. Right y axis: Eigensolve operation time for double precision arithmetic.

CPU runs. All in all, the performance is much worse than in the matrix-vector product case, as expected.

5 Conclusions

Computing a few eigenpairs of a block-tridiagonal matrix (or matrix pencil) is a computational problem that arises in different applications, e.g. in certain configurations of magnetohydrodynamic equilibrium solvers in the field of plasma physics [6]. We have explored GPU implementations in the context of the SLEPc library to compute either exterior or interior eigenvalues. In the former case, the main computational kernel is the matrix-vector product, that can be implemented with high Gflops rate. In the latter, it is necessary to perform a factorization of the matrix and then linear solves in each iteration of the eigensolver. The cyclic reduction algorithm on the GPU yields good performance for the factorization, but poorer in the case of triangular solves, as expected. Other algorithms for block-tridiagonal solves, such as those based on prefix sums, could be worth investigating to try to harness the full capability of the GPU.

In order to be able to address larger problems by exploiting several GPUs, either in the same node or in a compute cluster, we are currently developing MPI-based multi-GPU versions of the kernels, where each of the p MPI processes stores part of T and the local computations are carried out in the GPU. Hybrid CPU-GPU version could also be of interest. Also as a future work, we must optimize the orthogonalization of vectors on the GPU, since this cost becomes dominant once the time associated with the rest of operations has been reduced.

Acknowledgements The authors are grateful for the computing resources provided by the Spanish Supercomputing Network (RES).

References

1. Baghapour, B., Esfahanian, V., Torabzadeh, M., Darian, H.M.: A discontinuous Galerkin method with block cyclic reduction solver for simulating compressible flows on GPUs. Int. J. Comput. Math. 92(1), 110–131 (2014)
2. Bientinesi, P., Igual, F.D., Kressner, D., Petschow, M., Quintana-Ortí, E.S.: Condensed forms for the symmetric eigenvalue problem on multi-threaded architectures. Concur. Comput.: Pract. Exp. 23, 694–707 (2011)
3. Haidar, A., Ltaief, H., Dongarra, J.: Toward a high performance tile divide and conquer algorithm for the dense symmetric eigenvalue problem. SIAM J. Sci. Comput. 34(6), C249–C274 (2012)
4. Heller, D.: Some aspects of the cyclic reduction algorithm for block tridiagonal linear systems. SIAM J. Numer. Anal. 13(4), 484–496 (1976)
5. Hernandez, V., Roman, J.E., Vidal, V.: SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. ACM Trans. Math. Software 31(3), 351–362 (2005)
6. Hirshman, S.P., Perumalla, K.S., Lynch, V.E., Sanchez, R.: BCYCLIC: A parallel block tridiagonal matrix cyclic solver. J. Comput. Phys. 229(18), 6392–6404 (2010)
7. Minden, V., Smith, B., Knepley, M.G.: Preliminary implementation of PETSc using GPUs. In: GPU Solutions to Multi-scale Problems in Science and Engineering. pp. 1–9. Springer (2013)
8. NVIDIA: CUBLAS Library V7.0. Tech. Rep. DU-06702-001_v7.0, NVIDIA Corporation (2015)
9. Park, A.J., Perumalla, K.S.: Efficient heterogeneous execution on large multicore and accelerator platforms: Case study using a block tridiagonal solver. J. Parallel and Distrib. Comput. 73(12), 1578–1591 (2013)
10. Reguly, I., Giles, M.: Efficient sparse matrix-vector multiplication on cache-based GPUs. In: Innovative Parallel Computing (InPar). pp. 1–12 (2012)
11. Roman, J.E., Vasconcelos, P.B.: Harnessing GPU power from high-level libraries: eigenvalues of integral operators with SLEPc. In: International Conference on Computational Science. Procedia Comp. Sci., vol. 18, pp. 2591–2594. Elsevier (2013)
12. Seal, S.K., Perumalla, K.S., Hirshman, S.P.: Revisiting parallel cyclic reduction and parallel prefix-based algorithms for block tridiagonal systems of equations. J. Parallel and Distrib. Comput. 73(2), 273–280 (2013)
13. Stewart, G.W.: A Krylov–Schur algorithm for large eigenproblems. SIAM J. Matrix Anal. Appl. 23(3), 601–614 (2001)
14. Tomov, S., Nath, R., Dongarra, J.: Accelerating the reduction to upper Hessenberg, tridiagonal, and bidiagonal forms through hybrid GPU-based computing. Parallel Comput. 36(12), 645–654 (2010)
15. Vomel, C., Tomov, S., Dongarra, J.: Divide and conquer on hybrid GPU-accelerated multicore systems. SIAM J. Sci. Comput. 34(2), C70–C82 (2012)
16. Zhang, Y., Cohen, J., Owens, J.D.: Fast tridiagonal solvers on the GPU. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 127–136. PPoPP '10 (2010)