Martin Köhler Jens Saak

On GPU Acceleration of Common Solvers for (Quasi-) Triangular Generalized Lyapunov Equations

MPIMD/14-17 October 15, 2014

Abstract

The solutions of Lyapunov and generalized Lyapunov equations are a key player in many applications in systems and control theory. Their stable numerical computation, when the full solution is sought, is considered solved since the seminal work of Bartels and Stewart [1]. A number of variants of their algorithm have been proposed, but none of them goes beyond BLAS level-2 style implementation. On modern computers, however, the formulation of BLAS level-3 type implementations is crucial to enable optimal usage of cache hierarchies and modern block scheduling methods based on directed acyclic graphs describing the interdependence of single block computations. In this contribution, we present the port of our recent BLAS level-3 algorithm [9] to a GPU accelerator device.

Imprint:

Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg

Publisher:
Max Planck Institute for
Dynamics of Complex Technical Systems

Address:
Max Planck Institute for
Dynamics of Complex Technical Systems
Sandtorstr. 1
39106 Magdeburg

http://www.mpi-magdeburg.mpg.de/preprints/

Contents

1 Introduction

Lyapunov equations play an important role in systems and control theory. They are, e.g., a key ingredient in model order reduction, like Balanced Truncation [11], or part of the Newton Method for the Algebraic Riccati Equation [7]. Nowadays many practical situations require the solution of the generalized continuous-time Lyapunov equation

$$A^T X E + E^T X A = Y, \tag{1}$$

where A, E, X and Y are real $n \times n$ matrices. Furthermore we assume a symmetric right hand side Y such that the solution X is symmetric [13].

Before we derive a GPU based implementation we discuss some basics about the solution of generalized Lyapunov equations here. First, one can show that Equation (1) is uniquely solvable if and only if $\lambda_i + \lambda_j \neq 0$ for any two eigenvalues λ_i and λ_j of (A, E) [13, 4]. As a direct consequence, we know that if one of the matrices A or E is singular the Lyapunov equation is singular as well. The second consideration is that Equation (1) does not have any structure which is useful in order to develop a forward or backward substitution scheme. Also, the equivalent Kronecker product formulation of Equation (1)

$$\left(E^T \otimes A^T + A^T \otimes E^T\right) \operatorname{vec}(X) = \operatorname{vec}(Y),$$

where $\otimes$ denotes the Kronecker product of two matrices and $\operatorname{vec}(\cdot)$ the column-wise concatenation of an $n \times m$ matrix to a vector of length nm, does not deliver any further information. Like in the original Bartels-Stewart algorithm [1], we have to transform the equation before. Due to the generalized structure of the equation Penzl [13] proposed to use two orthogonal matrices Q and Z which transform the equation to generalized Schur form. This can be done for example using the QZ algorithm [10]. Applying these two transformation matrices we get

$$\begin{aligned} A &= Q^T A_s Z, \\ E &= Q^T E_s Z, \end{aligned} \tag{2}$$

where A_s is a (quasi-) upper triangular matrix and E_s is an upper triangular matrix. If we now plug the relation (2) into the Lyapunov equation (1) we end up with

$$\begin{aligned} Z^T A_s^T Q X Q^T E_s Z + Z^T E_s^T Q X Q^T A_s Z &= Y \\ A_s^T \underbrace{Q X Q^T}_{X_s} E_s + E_s^T \underbrace{Q X Q^T}_{X_s} A_s &= \underbrace{Z^T Y Z}_{Y_s}. \end{aligned} \tag{3}$$

On the one hand, this equation is equivalent to the original Equation (1) and the solution X is restored by

$$X = Q^T X_s Q. \tag{4}$$

On the other hand, it has an improved structure which is exploited for the formulation of a forward or backward substitution scheme.

In the remaining parts of this contribution, we only focus on Lyapunov equation which have already the structure shown in Equation (3). The acceleration of the matrix-matrix products necessary to get the solution of the original Lyapunov equation (1) are trivial using common software libraries like BLAS on the host and CUBLAS or clBLAS on the device. The only crucial operation is the computation of the transformation matrices Q and Z via the QZ algorithm which we assume done here. A multi core enabled way to compute them can, e.g., be found in [3, 2].

2 Blocked Variant of the Bartels-Stewart Algorithm

In addition to the (quasi-) upper triangular structure of the Lyapunov equation (3) we recall Penzl's extension [13] to the Bartels-Stewart algorithm [1], first. Employing this, we briefly discuss the key ideas to get a blocked algorithm out of it. The details of this blocked algorithm can be found in [9].

Regarding the structure of Equation (3), we partition the matrices A_s, E_s, Y_s and X_s into blocks depending on the eigenvalues of (A_s, E_s). If the diagonal entry of A_s belongs to a real eigenvalue the corresponding block is of size 1×1 and if the diagonal entry of a belongs to a complex conjugate eigenvalue pair the resulting block is of size 2×2. In this way, we get the following $p \times p$ block partitioning of (3):

$$\begin{aligned} A_s &= \begin{pmatrix} A_{11} & \cdots & A_{1p} \\ & \ddots & \vdots \\ 0 & & A_{pp} \end{pmatrix}, \quad E_s = \begin{pmatrix} E_{11} & \cdots & E_{1p} \\ & \ddots & \vdots \\ 0 & & E_{pp} \end{pmatrix}, \\ X_s &= \begin{pmatrix} X_{11} & \cdots & X_{1p} \\ \vdots & \ddots & \vdots \\ X_{p1} & \cdots & X_{pp} \end{pmatrix}, \quad Y_s = \begin{pmatrix} Y_{11} & \cdots & Y_{1p} \\ \vdots & \ddots & \vdots \\ Y_{p1} & \cdots & Y_{pp} \end{pmatrix}. \end{aligned} \tag{5}$$

Employing this block structure the solution of the Lyapunov equation (3) now gets equal to solving Sylvester equations

$$A_{kk}^T X_{kl} E_{ll} + E_{kk}^T X_{kl} A_{ll} = \hat{Y}_{kl} \tag{6}$$

Algorithm 1 Forward-Substitution for the generalized Lyapunov equation

```
Input: $(A_s, E_s)$ and $Y_s$ partitioned in $P_B$ blocks of size $N_B$, like in (5)
Output: $X_s$ solving the (quasi-) triangular Lyapunov equation (3)
 1: $X_s := Y_s$
 2: for $k = 1, \dots, P_B$ do
 3:   if $k > 1$ then
 4:     $X_{k,1:k-1} := X^T_{1:k-1,k}$ {Copy the symmetric part.}
 5:   end if
 6:   for $l = k, \dots, P_B$ do
 7:     if $l > 1$ then
 8:       $X_{k:l,l} := X_{k:l,l} - A^T_{k,k:l} X_{k,1:l-1} E_{1:l-1,l}$
 9:       $X_{k:l,l} := X_{k:l,l} - E^T_{k,k:l} X_{k,1:l-1} A_{1:l-1,l}$
10:     end if
11:     Solve $A^T_{k,k} X_* E_{l,l} + E^T_{k,k} X_* A_{l,l} = X_{k,l}$
12:     $X_{k,l} := X_*$
13:     if $k < l$ then
14:       $X_{k+1:l,l} := X_{k+1:l,l} - A^T_{k,k+1:l} X_{k,l} E_{l,l}$
15:       $X_{k+1:l,l} := X_{k+1:l,l} - E^T_{k,k+1:l} X_{k,l} A_{l,l}$
16:     end if
17:   end for
18: end for
```

with updated right hand sides $\hat{Y}_{kl}$:

$$\hat{Y}_{kl} = Y_{kl} - \sum_{\substack{i=1,j=1 \\ (i,j)\neq(k,l)}}^{k,l} \left(A^T_{ik} X_{ij} E_{jl} + E^T_{ik} X_{ij} A_{jl} \right) \tag{7}$$

for each block X_{kl}. Due to the fact that we assume to have a symmetric right hand side Y and respectively Y_s, the solution X_s will be symmetric as well. This allows us to only solve for $\frac{1}{2}p\,(p+1)$ blocks above or below the diagonal instead of solving for all p^2 blocks. An efficiency improving rearrangement of the right hand side update (7) is described in [13] and [9]. The remaining Sylvester equations (6) are reformulated to the corresponding Kronecker-Product representation [13, 15]:

$$\left(E^T_{ll} \otimes A^T_{kk} + A^T_{ll} \otimes E^T_{kk} \right) \operatorname{vec}(X_{kl}) = \operatorname{vec}\left(\hat{Y}_{kl} \right), \tag{8}$$

which are linear systems of size 1, 2 or 4 depending on A_{ll} and A_{kk}. If we solve for the upper triangle of X_s in a row-wise order we end up with Algorithm 1, which uses a block size of 1 or 2. Because the right hand side Y_{kl} is not longer required after we have solved for the corresponding X_{kl} we overwrite the right hand side Y_s with the solution X_s and in this way we get an in-place algorithm. The algorithm has an overall flop count of $\frac{8}{3}n^3 + \mathcal{O}\left(n^2\right)$ [13, 9].

As long as we only have the 1×1 or 2×2 block partitioning of Equation (3) the algorithm will only involve level-2 BLAS operations or matrix-matrix products of small size. This conflicts with the common guidelines for the development of efficient algorithms on current hardware architectures regardless of whether it is a CPU or GPU platform. If we now take a look at all operations performed in Algorithm 1 we see that all the updates on the right hand side work with arbitrary block sizes N_B as long as the matrix dimensions are compatible. From this point of view we can easily get to a blocked algorithm if we increase the size of the blocks in Equation (5). The only caveat is that we must not split the 2×2 blocks in the diagonal of A_s. That means in the partitioning of A_s we have to increase or decrease the block size by 1 if the partitioning with the original block size would lead to a splitting of a complex pair of eigenvalues.

However, if we increase the block size to get better performing right hand side updates we also increase the size of the inner Sylvester equation (6) and its Kronecker representation (8). If we assume a block size N_B for the inner Sylvester equation (6) the corresponding Kronecker formulation is of dimension $N_B^2 \times N_B^2$ and one LU decomposition of it costs $\frac{2}{3}N_B^6$ flops which is too much even for moderate block sizes, like $N_B = 64$. In [9] we discussed different variants to solve the inner Sylvester equations in $\mathcal{O}\left(N_B^3\right)$ and how they influence the overall flop count of Algorithm 1. The following paragraphs will introduce the fastest and most accurate approach from [9] to solve the inner Sylvester equation (6). The approach is based on the ideas of Gardiner et al. [5] and is adjusted to the structure of our equations.

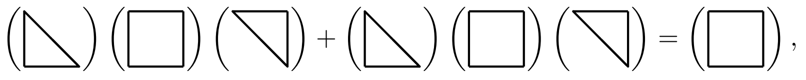
Modified Gardiner-Laub Approach to solve the inner Sylvester equations For easier reading, we denote the inner Sylvester equation (6)

$$A_{kk}^T X_{kl} E_{ll} + E_{kk}^T X_{kl} A_{ll} = \hat{Y}_{kl}$$

as

$$\hat{A}^T \hat{X} \hat{B} + \hat{C}^T \hat{X} \hat{D} = \hat{Y}, \tag{9}$$

where $\hat{A}$, $\hat{C} \in \mathbb{R}^{\hat{n} \times \hat{n}}$, $\hat{B}$, $\hat{D} \in \mathbb{R}^{\hat{m} \times \hat{m}}$ and $\hat{X}$, $\hat{Y} \in \mathbb{R}^{\hat{n} \times \hat{m}}$. Our application in Algorithm 1 and the Transformation (2) guarantee that this equation has the following structure:



where $\hat{A}$ and $\hat{D}$ may have 2×2 diagonal blocks depending on the eigenvalue structure of (A_s, E_s). Employing this structure allows us to rewrite the k-th column of $\hat{Y}$ as:

$$\hat{A}^T \sum_{l=1}^{k} \hat{B}_{lk} \hat{X}_{\cdot l} + \hat{C}^T \sum_{l=1}^{k+1} \hat{D}_{lk} \hat{X}_{\cdot l} = \hat{Y}_{\cdot k} \quad \text{for} \quad k = 1, \ldots, \hat{m}. \tag{10}$$

Depending on the diagonal entries of $\hat{D}$ we consider two cases. First, we assume that $\hat{D}_{k+1,k} = 0$, i.e., the diagonal entry of $\hat{D}$ belongs to a real eigenvalue. In this case, we can express the k-th column of the solution $\hat{X}$ as:

$$\begin{aligned}\hat{B}_{kk}\hat{A}^T\hat{X}_{\cdot k} + \hat{A}^T\sum_{l=1}^{k-1}\hat{B}_{lk}\hat{X}_{\cdot l} + \hat{D}_{kk}\hat{C}^T\hat{X}_{\cdot k} + \hat{C}^T\sum_{l=1}^{k-1}\hat{D}_{lk}\hat{X}_{\cdot l} &= \hat{Y}_{\cdot k} \\ \left(\hat{B}_{kk}\hat{A} + \hat{D}_{kk}\hat{C}\right)^T\hat{X}_{\cdot k} = \hat{Y}_{\cdot k} - \hat{A}^T\sum_{l=1}^{k-1}\hat{B}_{lk}\hat{X}_{\cdot l} - \hat{C}^T\sum_{l=1}^{k-1}\hat{D}_{lk}\hat{X}_{\cdot l}. &\qquad (11)\end{aligned}$$

Second, we consider the case where $\hat{D}_{k+1,k} \neq 0$. In this case, the diagonal entry of $\hat{D}$ belongs to a complex conjugate eigenvalue pair. Now, the k-th and $(k+1)$-th column of the solution $\hat{X}$ depend on each other and can be written as:

$$\begin{aligned}&\left(\hat{B}_{kk}\hat{A} + \hat{D}_{kk}\hat{C}\right)^T\hat{X}_{\cdot k} + \hat{D}_{k+1,k}\hat{C}^T\hat{X}_{\cdot k+1} \\ &\qquad = \hat{Y}_{\cdot k} - \hat{A}^T\sum_{l=1}^{k-1}\hat{B}_{lk}\hat{X}_{\cdot l} - \hat{C}^T\sum_{l=1}^{k-1}\hat{D}_{lk}\hat{X}_{\cdot l} = \bar{Y}_{\cdot k}\end{aligned}$$

and

$$\begin{aligned}&\left(\hat{B}_{k,k+1}\hat{A} + \hat{D}_{k,k+1}\hat{C}\right)^T\hat{X}_{\cdot k} + \left(\hat{B}_{k+1,k+1}\hat{A} + \hat{D}_{k+1,k+1}\hat{C}\right)^T\hat{X}_{\cdot k+1} \\ &\qquad = \hat{Y}_{\cdot k+1} - \hat{A}^T\sum_{l=1}^{k-1}\hat{B}_{lk}\hat{X}_{\cdot l} - \hat{C}^T\sum_{l=1}^{k-1}\hat{D}_{lk}\hat{X}_{\cdot l} = \bar{Y}_{\cdot k+1}.\end{aligned}$$

Rewriting this equation into a linear system of size $2\hat{n} \times 2\hat{n}$

$$\begin{pmatrix}\hat{B}_{kk}\hat{A} + \hat{D}_{kk}\hat{C} & \hat{B}_{k,k+1}\hat{A} + \hat{D}_{k,k+1}\hat{C} \\ \hat{D}_{k+1,k}\hat{C} & \hat{B}_{k+1,k+1}\hat{A} + \hat{D}_{k+1,k+1}\hat{C}\end{pmatrix}^T\begin{pmatrix}\hat{X}_{\cdot k} \\ \hat{X}_{\cdot k+1}\end{pmatrix} = \begin{pmatrix}\bar{Y}_{\cdot k} \\ \bar{Y}_{\cdot k+1}\end{pmatrix} \qquad (12)$$

we can solve for $\hat{X}_k$ and $\hat{X}_{k+1}$ at once. In this way, we determine all columns of the solution $\hat{X}$ without setting up the Kronecker-Product representation of the Sylvester equation. The whole procedure is shown in Algorithm 2. The algorithm needs $4\hat{n}^2+2\hat{n}$ floating point numbers extra memory and have a flop count of $10\hat{n}^3 - 2\hat{n}^2$ in the best case and $13\hat{n}^3 + \frac{31}{4}\hat{n}^2$ in the worst case [9]. In the context of the Lyapunov solver that means that as long as we use a moderate block size $N_B = \hat{n} = \hat{m}$ the flop count of the inner Sylvester equation solver does not influence the asymptotic flop count for the Bartels-Stewart algorithm in a mentionable way. There exists another approach to solve the inner Sylvester equation (9) based on the ideas by Kågström and Westin [8] which turned out to be slower and less accurate than the variant based on the idea by Gardiner and Laub [9].

Handling the inner Sylvester equations using Algorithm 2 we are able to use the Bartels-Stewart algorithm with level-3 BLAS operations for the right hand side updates. This already gains a considerable speed up on common desktop CPUs [9]. The only crucial point is that if we solve for a diagonal block X_{kk} we have to ensure its

Algorithm 2 Solution of the generalized Sylvester equation (9)

```
Input: $(A, C) \in \mathbb{R}^{n\times n}$ and $(D, B) \in \mathbb{R}^{m\times m}$ in real generalized Schur form, $Y \in \mathbb{R}^{n\times m}$.
Output: $X \in \mathbb{R}^{n\times m}$ solving $A^TXB + C^TXD = Y$
 1: $k := 1$
 2: while $k \leq n$ do
 3:    if $D_{k+1,k} = 0$ then
 4:       Solve (11) for $X_{\cdot k}$
 5:       $x_1 := A^TX_{\cdot k}, \quad x_2 := C^TX_{\cdot k}$
 6:       for $l = k+1, \dots, m$ do
 7:          $Y_{\cdot l} := Y_{\cdot l} - B_{k,l}x_1 - D_{k,l}x_2$
 8:       end for
 9:       $k := k + 1$
10:    else
11:       Solve (12) for $X_{\cdot k}$ and $X_{\cdot k+1}$
12:       $x_1 := A^TX_{\cdot k}, \quad x_2 := C^TX_{\cdot k}$
13:       $y_1 := A^TX_{\cdot k+1}, \quad y_2 := C^TX_{\cdot k+1}$
14:       for $l = k+2, \dots, m$ do
15:          $Y_{\cdot l} := Y_{\cdot l} - B_{k,l}x_1 - D_{k,l}x_2$
16:          $Y_{\cdot l} := Y_{\cdot l} - B_{k+1,l}y_1 - D_{k+1,l}y_2$
17:       end for
18:       $k := k + 2$
19:    end if
20: end while
```

symmetry. We proposed different strategies for this in [9], but we restrict the resymmetrization of the diagonal blocks using

$$\frac{1}{2}\left(X_{kk} + X_{kk}^T\right) \rightarrow X_{kk}, \tag{13}$$

here.

3 GPU Implementation

For the GPU implementation we use a hybrid GPU-CPU based approach similar to the approaches used by MAGMA [17, 16]. To this end, we split Algorithm 1 into two parts. The first one, which only involves level-3 BLAS operations, is the update of the right hand side. The second one is the solution of the inner Sylvester equation (6) which is solved with Algorithm 2. The basic idea is now to move all level-3 BLAS operations to the GPU and solve the remaining inner Sylvester equations on the CPU because Algorithm 2 involves only level-1 and level-2 BLAS operations. An easy implementation works as follows: We copy all three matrices A_s, E_s and Y_s to the device and we surround Step 11 of Algorithm 1 by transferring the current block $X_{k,l}$ from the device, solve the inner Sylvester equation on the host and $X_{k,l}$ back to the device. This is only the basic idea to use the GPU, but does not yet include any efficiency improving techniques like parallel computations on host and device or communication hiding via asynchronous data transfers. Rearranging Algorithm 1 in order to use these techniques (among others) is covered by the following paragraphs.

In order to enable our algorithm to make use of asynchronous operations we first have to figure out which data is necessary to compute the next iterate on the host such that we can prepare the data for the next iteration step while the GPU still finishes the current computations. A closer look to the order in which the blocks of X_s are computed (without the copy operations for the symmetric parts) we see the following scheme:

$$\begin{pmatrix} 1 & 2 & \cdots & p \\ & p+1 & \cdots & p+p-1 \\ & & \ddots & \vdots \\ & & & \frac{1}{2}p(p+1) \end{pmatrix}$$

That means for a computation scheme, that enables both GPU and CPU to work in parallel, we have to prepare the next block $X_{k,l+1}$ in a row before we perform the remaining work on the GPU. For a fixed row k the first right hand side update in Steps 8 and 9 of Algorithm 1 only requires data from the previous blocks in the current row. That means the update of $X_{k,l+1}$ depends only on the knowledge of $X_{k,1:l}$. If we now have solved for $X_{k,l}$ we can perform these updates. The problem is that the rest of the first right hand side update in column l overlaps with the second right hand side update in Steps 14 and 15 of Algorithm 1. This problem can be solved if we rearrange the Algorithm 1 such that we solve the small Sylvester equation for X_{kl} on the CPU first and then compute

$$\begin{aligned} X_{k:l+1,l+1} &:= X_{k:l+1,l+1} - A^T_{k,k:l+1} X_{k,1:l} E_{1:l,l+1} \\ X_{k:l+1,l+1} &:= X_{k:l+1,l+1} - E^T_{k,k:l+1} X_{k,1:l} A_{1:l,l+1} \end{aligned}$$

and

$$\begin{aligned} X_{k+1:l,l} &:= X_{k+1:l,l} - A^T_{k,k+1:l} X_{k,l} E_{l,l} \\ X_{k+1:l,l} &:= X_{k+1:l,l} - E^T_{k,k+1:l} X_{k,l} A_{l,l} \end{aligned}$$

in parallel. The results are the data access patterns for X_s shown in Figures 1 and 2 for both parts of the right hand side update. We easily see from the figures that the areas which are updated (shown on the left side of the arrow) are independent from each other with respect to write operations. This enables us to perform both updates independently. The only problem is that this only works for a fixed row. After one row is completed we have to copy the now known parts of the solution to their symmetric positions in the matrix X_s. At this point we have to synchronize all computations on the GPU and the CPU to prevent race conditions.

The described rearrangement of Algorithm 1 including the necessary data transfers is shown in Algorithm 3. The computations in Step 13 and the back transfer of the next right hand side $X_{k,l+1}$ in Step 14 are done in parallel to the remaining update of the

$$\begin{array}{c|c}
 & \begin{array}{cccc} & l & l+1 & \end{array} \\
\begin{array}{c} \\ k \\ \\ l+1 \\ \\ \end{array} & \begin{pmatrix} * & * & * & * \\ * & * & X_{k,l+1} & * \\ * & * & \vdots & * \\ * & * & X_{l+1,l+1} & * \\ * & * & * & * \end{pmatrix}
\end{array}
\leftarrow
\begin{array}{cc}
 & \begin{array}{cccc} 1 & & l & \end{array} \\
\begin{array}{c} \\ k \\ \\ \\ \\ \end{array} & \begin{pmatrix} * & * & * & * \\ X_{k,1} & \cdots & X_{k,l} & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix}
\end{array}$$

Figure 1: Access Pattern of X_s in the (k, l) iteration for the next right hand side update to compute $X_{k,l+1}$.

$$\begin{array}{cc}
 & \begin{array}{cccc} & l & l+1 & \end{array} \\
\begin{array}{c} \\ k+1 \\ \\ l \\ \\ \end{array} & \begin{pmatrix} * & * & * & * \\ * & X_{k+1,l} & * & * \\ * & \vdots & * & * \\ * & X_{k+1,l} & * & * \\ * & * & * & * \end{pmatrix}
\end{array}
\leftarrow
\begin{array}{cc}
 & \begin{array}{cccc} 1 & & l & \end{array} \\
\begin{array}{c} \\ k \\ \\ \\ \\ \end{array} & \begin{pmatrix} * & * & * & * \\ * & * & X_{k,l} & * \\ * & * & * & * \\ * & * & * & * \\ * & * & * & * \end{pmatrix}
\end{array}$$

Figure 2: Access Pattern of X_s in the (k, l) iteration for the second right hand side update.

current column in Step 17. On the Nvidia® CUDA architecture this behaviour can be implemented using streams and asynchronous communication [12]. Additionally after we have transfered the next block $X_{k,l+1}$ back to the host we solve the small Sylvester equation for that right hand side. The only necessary synchronization is after we finished a row of X_s. Beside this parallelization aspects the algorithm assumes that A_s, E_s and X_s reside on the CPU as well. This saves many transfer operations for copying the corresponding A_{ll}, E_{ll}, A_{kk} and E_{kk} back to the CPU when they are needed there.

Alongside these implementation aspects there exists a less obvious problem if we want to run Algorithm 3 efficiently on a GPU. One basic requirement of a fast GPU implementation is an ordered data access [12]. This means that the data, which is involved in a computation, should start at a cache-line boundary. Only if this condition is fulfilled the GPU can deploy its computational power as well as the full memory bandwidth. This is achieved using an even leading dimension for the matrix storage which is a multiple of the cache-line length,i.e., typically 128 byte. For many devices a multiple of 32 floating point numbers is good value because this fits to the cache-line length in single as well as in double precision. This condition is easily satisfied by rearranging the data layout when the matrices are copied to the device.

Because this property must especially hold for each block in A_s, E_s and X_s the block size N_B must be a multiple of the cache-line length as well. This is necessary such that each block operation gains an advantage out of the ordered memory access. But as we mentioned in Section 2 the block size N_B can differ by ± 1 with respect to the eigenvalue value distribution on the diagonal of A_s. If we consider the following

Algorithm 3 Forward-Substitution for the generalized Lyapunov equation on a GPU

```
Input: $(A_s, E_s)$ and $Y_s$ partitioned in $P_B$ blocks of size $N_B$, like in (5)
Output: $X_s$ solving the (quasi-) triangular Lyapunov equation (3)
 1: $X_s := Y_s$ and copy $A_s$, $E_s$ and $X_s$ to the device.
 2: for $k = 1, \dots, P_B$ do
 3:     for $l = k, \dots, P_B$ do
 4:         if $l > 1$ and $k = l$ then
 5:             Synchronize all computations.
 6:             $X_{k,1:k-1} := X^T_{1:k-1,k}$ on the GPU. {Copy the symmetric part.}
 7:             $X_{k,k} := X_{k,k} - A^T_{k,k} X_{k,1:k-1} E_{1:k-1,k} - E^T_{k,k} X_{k,1:k-1} A_{1:k-1,k}$ on the GPU.
 8:             Copy $X_{k,l}$ from the device to the host.
 9:         end if
10:         Solve $A^T_{k,k} X_* E_{l,l} + E^T_{k,k} X_* A_{l,l} = X_{k,l}$ on the host.
11:         $X_{k,l} := \frac{1}{2}\left(X_* - X_*^T\right)$ and upload it to the device.
12:         if $l < n$ then
13:             $X_{k:l+1,l+1} := X_{k:l+1,l+1} - A^T_{k,k:l+1} X_{k,1:l} E_{1:l,l+1} - E^T_{k,k:l+1} X_{k,1:l} A_{1:l,l+1}$ on the
                GPU.
14:             Copy $X_{k,l+1}$ from the device to the host.
15:         end if
16:         if $k < l$ then
17:             $X_{k+1:l,l} := X_{k+1:l,l} - A^T_{k,k+1:l} X_{k,l} E_{l,l} - E^T_{k,k+1:l} X_{k,l} A_{l,l}$ on the GPU.
18:         end if
19:     end for
20: end for
21: Synchronize all computations and copy $X_s$ from the device to the host.
```

example the problem becomes obvious: We assume a block size $N_B = 32$ which fulfills the requirements of the cache-line length and the $(32, 32)$ entry of the matrix A_s belongs to a complex eigenvalue pair. Then we have use either 31 or 33 as block size in this case. That means that all following blocks will be moved by one column and one row. The column shift does not disturb the alignment because all columns are aligned using a appropriate leading dimension but the row shift disturbs the memory access. Once we have at least one odd block size in the whole matrix partitioning the alignment for the remaining blocks in all three matrices A_s, E_s and X_s is destroyed.

One way to solve this problem is to reorder the eigenvalues of the pencil (A_s, E_s) such that all 2×2 diagonal block starts on an odd position. Using Givens rotations we can move the eigenvalues in (A_s, E_s) to any position on the diagonal [6]. Due to the fact that we do not want to introduce further restrictions to the block size, we sort all complex eigenvalue pairs to the upper left on the diagonal. Now, all 2×2 blocks start in an odd row and column position and we never get into trouble if the block size is a multiple of the cache-line length. The block structure of the matrix A_s in

n \ N_B	32	64	96	128
1 000	0.457	0.341	0.415	0.502
2 000	1.931	1.363	1.596	1.965
3 000	4.360	3.014	3.574	4.376
4 000	7.736	5.185	6.194	7.622
5 000	12.298	8.310	9.838	12.091
6 000	18.432	12.685	14.906	18.005

Table 1: Runtime (in s) for different block sizes N_B on the GPU with sorted eigenvalues.

Equation (5) then changes to

$$A_s := \begin{pmatrix} A_{1,1} & \cdots & A_{1,k} & a_{1,k+1} & \cdots & a_{1,p} \\ & \ddots & \vdots & \vdots & & \vdots \\ & & A_{k,k} & a_{k,k+1} & \cdots & a_{k,p} \\ & & & a_{k+1,k+1} & \cdots & a_{k+1,p} \\ & & & & \ddots & \vdots \\ & & & & & a_{p,p} \end{pmatrix}, \tag{14}$$

where $A_{*,*}$ denotes a 2×2 block and $a_{*,*}$ a scalar value. This reordering can be done using the QZ algorithm implemented in LAPACK. Using the additional *select* function as an argument one can select eigenvalues which should be moved to the upper left part of the matrix A_s. This gives us the desired structure presented in (14) and guarantees that all block operations in Algorithm 3 access the memory in the proper way. The same idea is used on the CPU in [9] to increase the performance using modern vector extension of the CPU.

We skip a multi-GPU implementation because of the expected high communication effort. The reason behind this is that none of the usual data distributions, like row-/column-wise block cyclic or block checkerboard layout fits to the computation scheme.

4 Numerical Results

In this section we present the performance and accuracy results to show that the GPU implementation provides comparable results as the current SLICOT [13] and the level-3 BLAS CPU [9] implementations. All computations are performed on a dual socket Intel® Xeon® E5-2690 (2×8 Cores, 2.9 GHz) equipped with 32 GB RAM. As accelerator device we use one Nvidia® Tesla K20m card. All test are performed under Ubuntu 14.04 using the Intel C and Fortran Compiler 14.0, the Intel® MKL 11.1 library for the host computations and Nvidia® CUDA 6.5 on the device. The host computations for the GPU variant are executed in single-thread mode because the level-2 BLAS code for the small Sylvester equations does not scale well on the 16 core architecture. The host implementation `SG03AY` from SLICOT [14] and its level-3

n	SG03AY (SLICOT [13])	SG03CY (level-3 BLAS [9])	GPU	Speed up relative to SLICOT	level-3 BLAS
1 000	1.744	0.312	0.341	5.120	0.917
2 000	21.119	1.392	1.363	15.494	1.021
3 000	57.166	3.368	3.014	18.970	1.118
4 000	135.156	6.063	5.185	26.066	1.169
5 000	245.516	10.116	8.310	29.544	1.217
6 000	426.566	15.491	12.685	33.627	1.221

Table 2: Runtime (in s) and speed up of the different implementations.

n	SG03AY (SLICOT)	SG03CY (level-3 BLAS)	GPU
1 000	1.319e-15	5.677e-16	8.299e-16
2 000	1.874e-15	5.997e-16	7.639e-16
3 000	2.213e-15	6.201e-16	6.110e-16
4 000	2.641e-15	6.912e-16	6.348e-16
5 000	2.800e-15	7.204e-16	6.363e-16
6 000	3.190e-15	7.811e-16	6.875e-16

Table 3: Average relative residual of the different implementations with optimal block size.

BLAS reformulation SG03CY [9] use the MKL with 16 threads in order to maximize exploitation of the host CPU features.

The numerical tests are performed with a set of random problems. We use scalable random matrix pencils to analyze the performance without focusing on a special eigenvalue structure which may influence the performance as described in Section 3. Each matrix pencil is generated via two subsequent calls of DLARNV from LAPACK. The initial seed for this subroutine is set to $(1, 1, 1, 1)$ and incremented internally during each call. For each test we generate ten of these pencils. The pencils are transformed to generalized Schur form before the benchmarks start. That means, we focus on the performance of solving Equation (3). In order to figure out the difference between sorted and unsorted eigenvalues on the diagonal of A_s we computed the QZ decomposition of (A, E) twice with and without the additional *select* function to order the eigenvalues. For all cases we compute the right hand side Y such that the true solution $X^{(true)}$ is the matrix with all unit entries.

Before we compare the GPU implementation in Algorithm 3 with the two existing CPU implementations, we have to determine the optimal block size N_B for Algorithm 3 first. Therefore we solve the Lyapunov equation (3) for the already reduced random matrix pencils with different block sizes. We use 32, 64, 96 and 128 in order to fit to the cache-line boundary problem described in Section 3. Preliminary tests showed that using a smaller or a larger block size will not lead to a performance improvement. The complex eigenvalue pairs are sorted to the upper left block of the matrix. In Table 1 we see that independent of the problem size the minimal runtime is achieved using a

	1 000	2 000	3 000	4 000	5 000	6 000	Average
Unorted ev.	0.388	1.399	3.177	5.764	9.024	13.562	
Sorted ev.	0.341	1.363	3.014	5.185	8.310	12.685	
Ratio	1.139	1.027	1.054	1.112	1.086	1.069	1.081

Table 4: Influence of the eigenvalue sorting on the runtime.

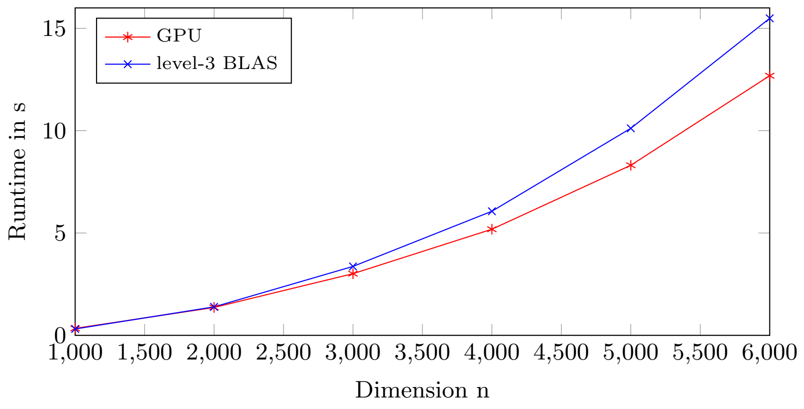
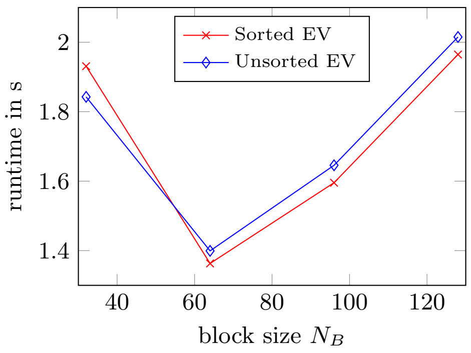


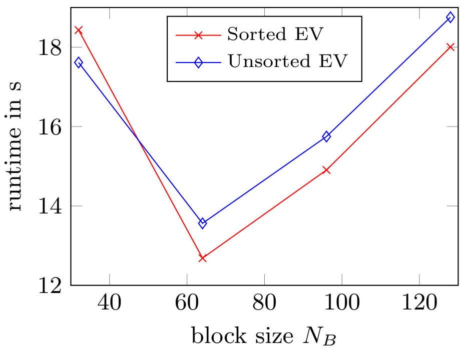
Figure 3: Runtime comparison of the GPU and level-3 BLAS CPU version.

block size of $N_B^{opt} = 64$ which we use for the remaining computations. In contrast to the block CPU implementation [9] the selection of a non optimal block sizes slows the algorithm down drastically. From the Tables 1 and 2 we see that even small deviations from the optimal block size will results in the same or even slower performance as the CPU implementation.

The second test is the comparison between the two existing CPU implementations, namely the old level-2 BLAS version from Penzl [13] and the level-3 BLAS presented in [9], against the new GPU implementation. Table 2 show that the GPU implementation gains a large speed up in direct comparison against the old level-2 BLAS implementation. Even with moderate size problems of dimension $n = 1\ 000$ the GPU implementation is much faster than the old one. In comparison to the level-3 implementation we only gain a performance improvement for large problems. From Table 2 and Figure 3 we see that the performance gain gets larger with increasing block size, but the problem must be at least of dimension $n = 2\ 000$ to beat the CPU implementation. There are two reasons behind this behavior. On the one hand, the CPU has a high peak performance of 371.2 GFlops/s in double precision. On the other hand, the size of the matrix products is not large enough to fully load the GPU. But from Table 1 we have already seen that increasing the block size will slow down the computations because then the communication between host and device will take more time and the level-2 BLAS solver for the inner Sylvester equation is not efficient for large block sizes.



(a) $n = 2\,000$



(b) $n = 6\,000$

Figure 4: Optimal block size with and without sorted eigenvalues.

Beside the performance we compare the relative residuals of all three implementations. From Table 3 we see that the GPU implementation produces comparable results in the same order of accuracy as the host implementations as we expect it for an IEEE compatible device.

In Section 3 we described that we can only get good performance of the GPU if we access the memory on the device in a well structured way. Therefore, we sorted all complex conjugate eigenvalue to the upper left part of the matrix A_s in the preparatory computations. If we do not sort the eigenvalues before, e.g., by deactivating the *select* subroutine in LAPACK's QZ algorithm, the performance of the GPU should decrease. In Table 4 we show the runtime of Algorithm 3 in the case of a sorted and a unsorted spectra. The average performance gain of using sorted eigenvalues is about 8%. Interestingly this is not valid if we use a block size of $N_B = 32$, where Figure 3 shows that using sorted eigenvalues is slower than using the sorted diagonal.

All benchmarks assume that the pencils is already transformed to (quasi-) upper triangular form. If this it not the case we have to add the overhead of computing the generalized Schur decomposition to the runtime. However, in situations, where the Lyapunov equation needs to be solved for different right hand sides and the Schur decomposition can be reused, and the overhead shrinks.

5 Conclusions

We have seen how we can build a GPU accelerated variant of the generalized Bartels-Stewart algorithm on top of the level-3 BLAS implementation in [9]. By reordering the computations in the algorithm we are able to use asynchronous communications and overlapping of the host and device computations. Our new implementation is 5 to 33 times faster than the current implementation in SLICOT [13] on which even the MATLAB `lyap` solver is based on. In case of the new level-3 BLAS implementation [9] we still get a performance gain of 20%. One reason behind this comparably small gain is the rather powerful host CPU on systems with less powerful CPUs a higher performance gain is possible. We have also seen that rearranging the input data to get a well ordered data access scheme will result in an additional speed up. The only remaining piece to get an GPU accelerated solver for the non-triangular Lyapunov equation (1) is the efficient computation of the generalized Schur decomposition which is part for future research.

References

[1] R. H. Bartels and G. W. Stewart, *Solution of the matrix equation $AX + XB = C$: Algorithm 432*, Comm. ACM, 15 (1972), pp. 820–826.

[2] P. Benner, M. Köhler, and J. Saak, *A cache-aware implementation of the spectral divide-and-conquer approach for the non-symmetric generalized eigenvalue problem*, Proc. Appl. Math. Mech., (2014). submitted.

[3] ——, *Fast approximate solution of the non-symmetric generalized eigenvalue problem on multicore architectures*, in Parallel Computing: Accelerating Computational Science and Engineering (CSE), M. Bader, A. B. H.-J. Bungartz, M. Gerndt, G. R. Joubert, and F. Peters, eds., vol. 25 of Advances in Parallel Computing, IOS Press, 2014, pp. 143–152.

[4] E. K.-W. Chu, *The solution of the matrix equations AXB - CXD=E and (YA - DZ,YC - BZ)=(E,F)*, Linear Algebra Appl., 93 (1987), pp. 93 – 105.

[5] J. D. Gardiner, A. J. Laub, J. J. Amato, and C. B. Moler, *Solution of the Sylvester matrix equation $AXB + CXD = E$*, ACM Trans. Math. Software, 18 (1992), pp. 223–231.

[6] G. H. Golub and C. F. Van Loan, *Matrix Computations*, Johns Hopkins University Press, Baltimore, third ed., 1996.

[7] S. J. Hammarling, *Newton's method for solving the algebraic Riccati equation*, NPL Report DITC 12/82, National Physical Laboratory, Teddington, Middlesex TW11 OLW, U.K., 1982.

[8] B. Kågström and L. Westin, *Generalized Schur methods with condition estimators for solving the generalized Sylvester equation*, IEEE Trans. Automat. Control, 34 (1989), pp. 745–751.

[9] M. Köhler and J. Saak, *On BLAS Level-3 Implementations of Common Solvers for (Quasi-) Triangular Generalized Lyapunov Equations*, SLICOT Working Note 2014-1, Sept. 2014. Available from `www.slicot.org`.

[10] C. B. Moler and G. W. Stewart, *An algorithm for generalized matrix eigenvalue problems*, SIAM J. Numer. Anal., 10 (1973), pp. 241–256.

[11] B. C. Moore, *Principal component analysis in linear systems: controllability, observability, and model reduction*, IEEE Trans. Automat. Control, AC-26 (1981), pp. 17–32.

[12] NVIDIA Corporation, *CUDA C Best Practices Guide.* http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.

[13] T. Penzl, *Numerical solution of generalized Lyapunov equations*, Adv. Comp. Math., 8 (1997), pp. 33–48.

[14] *SLICOT*, http://www.slicot.org.

[15] J. J. Sylvester, *The Collected Mathematical Papers of James Joseph Sylvester, Volume 4*, AMS Chelsea Publishing, 1973.

[16] S. Tomov, J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems.*, LAPACK Working Note 210, 2009.

[17] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, *Dense linear algebra solvers for multicore with GPU accelerators.*, LAPACK Working Note 225, 2010.