GPU ACCELERATION OF SPLITTING SCHEMES APPLIED TO DIFFERENTIAL MATRIX EQUATIONS

HERMANN MENA, LENA-MARIA PFURTSCHELLER, AND TONY STILLFJORD

Abstract. We consider differential Lyapunov and Riccati equations, and generalized versions thereof. Such equations arise in many different areas and are especially important within the field of optimal control. In order to approximate their solution, one may use several different kinds of numerical methods. Of these, splitting schemes are often a very competitive choice. In this article, we investigate the use of graphical processing units (GPUs) to parallelize such schemes and thereby further increase their effectiveness. According to our numerical experiments, speed-ups of several orders of magnitude are often observed for sufficiently large matrices. We also provide a comparison between different splitting strategies, demonstrating that splitting the equations into a moderate number of subproblems is generally optimal.

1. Introduction

We are interested in differential matrix equations of Lyapunov or Riccati type, or generalized versions of these. They are all of the form

$$\dot{P} = A^\mathsf{T} P + PA + Q + G(P),$$

where $A \in \mathbb{R}^{n\times n}$ and $Q \in \mathbb{R}^{n\times n}$ are given matrices and G is a matrix-valued function of P. For differential Lyapunov equations (DLE) we have $G(P) = 0$ and for differential Riccati equations (DRE) we have $G(P) = -PBR^{-1}B^\mathsf{T}P$ with two given matrices $B \in \mathbb{R}^{n\times m}$ and $R \in \mathbb{R}^{m\times m}$. Such equations occur frequently in many different areas, such as in optimal/robust control, optimal filtering, spectral factorizations, $\mathbf{H}_\infty$-control, differential games, etc. [1, 5, 27, 37].

Perhaps the most relevant setting is the linear quadratic regulator (LQR) problem. There, the aim is to optimize a finite-time cost function of the form

$$J(u) = \int_0^T \|y(t)\|^2 + \|u(t)\|^2 \mathrm{d}t, \quad T \geq 0,$$

under the constraints that $\dot{x} = Ax + Bu$ (state equation) and $y = Cx$ (output equation, with $C \in \mathbb{R}^{p\times n}$). In this case, the solution to the DLE with $Q = C^\mathsf{T}C$ gives the observability Gramian of the system, which characterizes the relevant states x for the input-output mapping $u \mapsto y$. The solution of the DRE, on the other hand, provides the optimal input that minimizes J, in state feedback form. In fact, if P solves the DRE with $Q = C^TC$ then the optimal input u^{opt} is given by $u^{\mathrm{opt}}(t) = -R^{-1}B^\mathsf{T}P(T-t)x(t)$.

For the generalized DLE and DRE versions, an additional linear term SPS^T appears in $G(P)$, where $S \in \mathbb{R}^{n\times n}$ is a given matrix. Such equations also arise in the LQR setting, when a stochastic perturbation of multiplicative type is included in the state equation.

Date: May, 22, 2018.

2010 *Mathematics Subject Classification* 65F30, 65Y05, 65F60.

*Keywords:*Differential Lyapunov equations, Differential Riccati equations, Large-scale, Splitting schemes, GPU acceleration.

In recent years, a number of numerical methods have been suggested for large-scale DLEs, DREs and related equations. The classic ones, low-rank versions of BDF and Rosenbrock schemes [14, 15, 30] are usually outperformed by more modern methods such as Krylov-based projection schemes [28], peer methods [29] or splitting schemes [31, 41, 42]. In this paper, we focus on splitting schemes. These methods lower the computational cost by dividing the problem into simpler subproblems such as $\dot{P} = A^\mathsf{T}P + PA$ and $\dot{P} = Q$ and then solve these separately, in sequence. While the splitting of course introduces an additional error, this is generally compensated by the decreased computational cost and leads to large speed-ups.

The hypothesis to be investigated in this paper is that utilizing a graphical processing unit (GPU) to parallelize the schemes may further greatly increase the efficiency. Such speed-ups have already been observed for other related methods for DREs [9, 10, 11] as well as for their steady-state versions; the algebraic Lyapunov and Riccati equations [11, 12]. In the just mentioned cases, the basic building block of the schemes is the computation of the matrix sign function, which requires the inversion of a large dense matrix. In a splitting scheme, the basic building block is instead the computation of the action of a matrix exponential on a skinny matrix. Speed-ups have previously been observed for applications where matrix exponentials are multiplied by vectors [3, 21], see also [22]. In these works, a speed-up is generally not observed for "small" matrices ($n \lesssim 1000$), and the speed-up is of limited size when the matrices are sparse rather than dense. As we are typically interested in at least medium-sized problems ($1000 \lesssim n \lesssim 10000$) we do expect to see a significant speed-up. Moreover, while we are necessarily considering the sparse case, we are not simply computing the action of the matrix exponential on vectors, but on skinny block matrices. This increases the parallelizability of the problem and makes the sparsity issues noted in e.g. [23, 7, 21] less relevant.

Since the relevant methods are mainly implemented in Matlab, we restrict ourselves to utilizing its built-in GPU support [38] via NVIDIA's CUDA [34] parallel programming interface. We do not claim that this approach leads to the best possible performance. The point is rather to demonstrate that quite simple changes to the implementations of the splitting schemes may lead to much better performance, when one has access to a GPU. Our results already show a remarkable improvement in efficiency, and this can only increase with further optimisations and the use of more advanced techniques tailored to specific problems.

In addition, we provide comparisons between different splitting strategies for DLEs and DREs. We particularly address questions that naturally arise while solving these equations by splitting methods. E.g., should the DLE be split at all? Should the DRE be split into two or three subproblems? Our results in this direction demonstrate that it is usually beneficial to use the smallest number of splits. However, when Q is sufficiently small it is beneficial to split it too, since the extra error is similarly small and the subproblems $\dot{P} = A^\mathsf{T}P + PA$ and $\dot{P} = Q$ are very cheap to compute compared to $\dot{P} = A^\mathsf{T}P + PA + Q$.

An outline of the article is as follows. In Section 2 we review the idea behind splitting schemes and apply them to all the mentioned equation types. Then we consider implementation details and the simple changes necessary for GPU utilization in Section 3. We present the results of several numerical experiments in Section 4, and summarize our conclusions in Section 5.

2. Splitting schemes

Splitting schemes are numerical methods that are applicable to differential equations that have a natural decomposition into two (or more) parts;

$$\dot{P} = F(P) = F_1(P) + F_2(P), \quad P(0) = P_0.$$

With "natural decomposition" we mean that the subproblems

$$\dot{P} = F_1(P) \quad \text{and} \quad \dot{P} = F_2(P)$$

are either simpler or cheaper to solve than the full problem $\dot{P} = F(P)$. This is the case in many problems, with the most common example being reaction-diffusion equations $\dot{x} = \Delta x + f(x)$. In this case, there are highly optimized methods for the pure diffusion problem $\dot{x} = \Delta x$, while the subproblem $\dot{x} = f(x)$ often turns into a local rather than global problem — i.e. it is enough to solve $\dot{x}_i = f(x_i)$ for every discretization point x_i. In the following, we denote the solution to $\dot{P} = F_k(P)$, $P(0) = P_0$, by $P(t) =: \mathcal{T}_k(t)P_0$.

The most basic and commonly used (exponential) splitting schemes are the Lie and Strang splittings. They are given by the time stepping operators

$$\mathcal{L}_h P_0 = \mathcal{T}_2(h)\,\mathcal{T}_1(h)\,P_0 \quad \text{and} \quad \mathcal{S}_h P_0 = \mathcal{T}_1\left(\frac{h}{2}\right)\mathcal{T}_2(h)\,\mathcal{T}_1\left(\frac{h}{2}\right)P_0,$$

respectively, where h is the time step. Of course, the roles of $\mathcal{T}_1$ and $\mathcal{T}_2$ might be interchanged. The schemes are then defined by

$$P_{k+1}^L = \mathcal{L}_h P_k^L \quad \text{and} \quad P_{k+1}^S = \mathcal{S}_h P_k^S,$$

with $P_0^L = P_0^S = P_0$. Here, P_k^L and P_k^S both approximate $P(kh)$. The Lie splitting is first-order accurate while Strang splitting is second-order accurate under certain conditions on F_1, F_2 and F, see e.g. [26]. For simplicity, we restrict ourselves to the Strang splitting scheme in this paper, but one might also consider higher-order schemes [20, 24, 42], or schemes where the subproblems are not solved exactly, see e.g. [26, 25].

Clearly, one might continue the splitting procedure if the system is naturally decomposed into more than two parts. If

$$\dot{P} = F_1(P) + F_2(P) + F_3(P)$$

then applying the Lie and Strang splitting schemes twice leads to the schemes

$$\begin{aligned}\tilde{\mathcal{L}}_h P_0 &= \mathcal{T}_3(h)\,\mathcal{T}_2(h)\,\mathcal{T}_1(h)\,P_0 \quad \text{and} \\ \tilde{\mathcal{S}}_h P_0 &= \mathcal{T}_1\left(\frac{h}{2}\right)\mathcal{T}_2\left(\frac{h}{2}\right)\mathcal{T}_3(h)\,\mathcal{T}_2\left(\frac{h}{2}\right)\mathcal{T}_1\left(\frac{h}{2}\right)P_0.\end{aligned}$$

Again, the roles of $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_3$ might be interchanged. Different compositions with a possibly higher number of operators might also be considered, in order to optimize the structure of the error. We refer to [4] but do not consider such methods here.

Like essentially every other method for solving differential matrix equations, the splitting schemes need to make use of low-rank structure in order to be competitive in the large-scale setting. This means that we can expect the singular values of the symmetric, positive semi-definite solution P to decay rapidly, see e.g. [2, 6, 8, 36, 40], and thus we can factorize $P \approx LDL^{\mathsf{T}}$ for $L \in \mathbb{R}^{n\times r}$, $D \in \mathbb{R}^{r\times r}$ with $r \ll n$. By formulating the methods to only operate on L and D and never explicitly form the product LDL^T, we drastically lower both the memory requirements and the computational cost.

In the following, we outline different splitting strategies for all the matrix equations mentioned so far, and also review how to low-rank-factorize each arising subproblem.

2.1. **Differential Lyapunov equations.** As a first example, we consider the differential Lyapunov equation

$$\dot{P} = A^\mathsf{T} P + PA + Q, \quad P(0) = P_0, \tag{1}$$

Here we may choose F_1 as the linear part and F_2 as the constant term, i.e.

$$F_1(P) = A^\mathsf{T} P + PA \quad \text{and} \quad F_2(P) = Q.$$

These subproblems can be solved explicitly and the solutions at time h are given by

$$\mathcal{T}_1(h)P_0 = e^{hA^\mathsf{T}} P_0 e^{hA},$$
$$\mathcal{T}_2(h)P_0 = P_0 + hQ.$$

It is easily seen that if we have the LDL^T-factorizations $P_0 = LDL^\mathsf{T}$ and $Q = L_Q D_Q L_Q^\mathsf{T}$, then we can also factorize these solutions as

$$\mathcal{T}_1(h)P_0 = \left(e^{hA^\mathsf{T}} L\right) D \left(e^{hA^\mathsf{T}} L\right)^\mathsf{T}, \tag{2}$$

$$\mathcal{T}_2(h)P_0 = \begin{bmatrix} L & L_Q \end{bmatrix} \begin{bmatrix} D & 0 \\ 0 & hD_Q \end{bmatrix} \begin{bmatrix} L & L_Q \end{bmatrix}^\mathsf{T}. \tag{3}$$

We could also note that the exact solution to the full problem is given by

$$P(t) = e^{tA^\mathsf{T}} P_0 e^{tA} + \int_0^t e^{sA^\mathsf{T}} Q e^{sA} ds, \quad t \in [0, T] \tag{4}$$

where the integral term may be approximated by high-order quadrature as in [41]. While this does not result in a splitting scheme of the form described above, we still include it in our experiments due to its similarity and efficiency.

2.2. **Differential Riccati equations.** A second example is given by the differential Riccati equation:

$$\dot{P} = A^\mathsf{T} P + PA + Q - PBR^{-1}B^\mathsf{T} P, \quad P(0) = P_0. \tag{5}$$

In this case, we can either split in three terms;

$$F_1(P) = A^\mathsf{T} P + PA, \quad F_2(P) = Q, \quad \text{and} \quad F_3(P) = PBR^{-1}B^\mathsf{T} P,$$

or two terms[1];

$$F_{12}(P) = A^\mathsf{T} P + PA + Q, \quad \text{and} \quad F_3(P) = PBR^{-1}B^\mathsf{T} P.$$

The latter was advocated in [41, 42] because (experimentally) the error constant in the three-term splitting is much larger. However, the three-term splitting does not need to approximate the integral term, and thus the larger error might be compensated by a lower computational cost.

In either case, we note that the solution at time h to the problem $\dot{P} = F_3(P)$, $P(0) = P_0$, is given explicitly by

$$\mathcal{T}_3(h)P_0 = (I + hP_0 BR^{-1}B^\mathsf{T})^{-1} P_0. \tag{6}$$

A low-rank factorization is given by

$$\mathcal{T}_3(h)LDL^\mathsf{T} = L(I + hDL^\mathsf{T} BR^{-1}B^\mathsf{T} L)^{-1} DL^\mathsf{T}.$$

Note that the I in this equation is not the same identity matrix as in the previous equation, because the DL^T-part of P_0 has moved. We thus only need to solve a small linear equation system.

[1]We deliberately use F_{12} and F_3 here rather than F_1 and F_2, in order to not change the meaning of the previously defined F_1 and F_2. The two-term splitting schemes are obviously still well-defined after substituting the proper numbers.

2.3. **Generalized Lyapunov equations.** We further consider a generalized Lyapunov equation of the form

$$\dot{P} = A^\mathsf{T} P + PA + Q + SPS^\mathsf{T}, \quad P(0) = P_0. \tag{7}$$

We again split the equation and obtain three subproblems defined by[2]

$$F_1(P) = A^\mathsf{T} P + PA, \quad F_2(P) = Q, \quad \text{and} \quad F_4(P) = SPS^\mathsf{T}.$$

The first two subproblems are handled as before, whereas we approximate $\mathcal{T}_4(h)(P)$ by the midpoint rule, analogously to what is done in [19]:

$$\mathcal{T}_4(h)P_0 \approx P_0 + hS\Big(P_0 + \frac{h}{2}SP_0S^\mathsf{T}\Big)S^\mathsf{T}.$$

Given $P_0 = L_0D_0L_0^\mathsf{T}$, we get $\mathcal{T}_4(h)P_0 \approx LDL^\mathsf{T}$, where

$$L = \left[L_0, \sqrt{h}SL_0, \frac{h}{\sqrt{2}}S^2L\right], \quad \text{and} \quad D = \texttt{blkdiag}(D_0, D_0, D_0),$$

where `blkdiag` is the block diagonal operator that puts its block arguments on the diagonal of an otherwise zero matrix.

We note that when using a second-order splitting scheme like the Strang splitting, it is necessary to use a second-order method like the midpoint rule in order to preserve the overall convergence order. If we use instead a first-order scheme like the Lie splitting, it is sufficient to approximate $\dot{P} = F_4(P)$ by e.g. the explicit Euler method.

2.4. **Generalized Riccati equations.** Moreover, we study a generalized Riccati equation given by

$$\dot{P} = A^\mathsf{T} P + PA + Q + SPS^\mathsf{T} - PBR^{-1}B^\mathsf{T}P, \quad P(0) = P_0, \tag{8}$$

and split this equation into three subproblems of the form

$$F_{12}(P) = A^\mathsf{T} P + PA + Q, \quad F_3(P) = -PBR^{-1}B^\mathsf{T}P, \quad \text{and} \quad F_4(P) = SPS^\mathsf{T}.$$

These subproblems are solved similarly as in the previous subsections. We do not consider a four term splitting since the extra error due to the splitting would become prohibitively large.

3. GPU considerations

In this section we briefly describe the GPU implementations of the Strang splitting applied to the differential matrix equations discussed in Section 2. All the algorithms are implemented in Matlab 2017a® using the Parallel Computing Toolbox. As there exist many built-in GPU-based functions, implementing algorithms on the GPU is quite user-friendly in recent releases of Matlab, see e.g. [38]. In order to avoid unnecessary communication between the CPU and the GPU, we first move all the data to the GPU, solve the equations on the GPU and transfer the results back to the CPU. These two steps are accomplished by the Matlab commands `gpuArray` and `gather`, respectively.

[2]For the same reason as in the previous note, we use F_4 rather than F_3 here.

3.1. **Action of the matrix exponential.** In all the considered equations, the most demanding part is to compute the action of the matrix exponential in (2) efficiently. In [17, 18] the authors considered an algorithm based on Leja interpolation and showed that applying the algorithm to a matrix derived from a spatial discretization of a differential operator is very efficient. We will thus use this method to compute $e^{hA}L$ for different skinny matrices L. In the following, we denote this algorithm by `expleja`.

We briefly sketch the implementation of the method and refer to [17, 18] for details. As a first step, the spectrum of the matrix A is estimated by the Gershgorin disk theorem. After moving A to the GPU by use of `gpuArray`, we take advantage of GPU-based built-in functions like `diag`, `sum` and `abs` and implement this algorithm equivalently to the CPU version. We proceed similarly with the computation of the parameters for the exponential interpolation, see [17]. The most time consuming part of the algorithm is the Newton interpolation that one has to perform at every Leja point. Here we profit from the efficient matrix-vector and matrix-matrix multiplication for large-scale matrices on the GPU, and the property that the multiplication of two GPU arrays is again stored on the GPU. Thus, the GPU implementation pays off particularly in this section of the code and we never have to copy the matrices from the GPU to the CPU and vice versa.

3.2. **Computing the matrix equations on the GPU.** In this subsection we will explain the methods used to solve the differential matrix equations on the GPU. First, we consider the DLE solved by the Strang splitting approach. We assume that the matrices are stored on the CPU, hence as a first step we copy these matrices to the GPU. Then, we can apply the steps from the CPU implementation as described in [42], computing the actions of the matrix exponential by Leja interpolation. A detailed description is given in Algorithm 3.1.

Algorithm 3.1 Solving DLE by Strang splitting on the GPU

1: Given: A, Q, P_0, T, N_t, $h = \frac{T}{N_t}$.
2: Compute LDL^T-decompositions of $Q = L_Q D_Q L_Q^\mathsf{T}$ and $P_0 = LDL^\mathsf{T}$ and copy matrices to GPU using `gpuArray`.
3: Compute parameters `param` for Leja interpolation.
4: **for** $k = 1 : N_t$ **do**
5: $\quad L = \texttt{expleja}(h/2, A, L, \texttt{param})$
6: $\quad L = [L, L_Q]$
7: $\quad D = \texttt{blkdiag}(D, hD_Q)$;
8: $\quad [L, D] = \texttt{column\_compression}(L, D)$;
9: $\quad L = \texttt{expleja}(h/2, A, L, \texttt{param})$
10: **end for**
11: $P = LDL^\mathsf{T}$;
12: $P = \texttt{gather}(P)$.

On the other hand, as mentioned in Subsection 2.1 it is possible to derive an explicit form of the solution of the DLE given by (4). Hence, following [42] we compute an approximation of the solution as given in Algorithm 3.2 using a quadrature rule to estimate the integral.

We note that in both Algorithm 3.1 and Algorithm 3.2 there is a so-called column compression step. This refers to the procedure of discarding (almost) linearly dependent columns from L, and serves to keep the number of columns in the approximations small. Without such a step, each iteration of Algorithm 3.1 (for example) would add the columns in L_Q to L, while the rank would likely stay similar. The

Algorithm 3.2 Solving DLE by Quadrature Rule on the GPU

1: Given: A, Q, P_0, T, N_t, $h = \frac{T}{N_t}$.
2: Repeat Steps 2 and 3 from Algorithm 3.1.
3: Approximate integral:
- Compute n nodes s_k and weights w_k of quadrature formula;
- $L_I = [\texttt{expleja}(s_1, A, L_Q), \ldots \texttt{expleja}(s_n, A, L_Q)]$;
- $D_I = \texttt{blkdiag}(w_1 D_Q, \ldots, w_n D_Q)$;
- $[L_I, D_I] = \texttt{column\_compression}(L_I, D_I)$.

4: **for** $k = 1, \ldots, N_t$ **do**
5: $L = [\texttt{expleja}(h, A, L, \texttt{param}), L_I]$
6: $D = \texttt{blkdiag}(D, D_I)$;
7: $[L, D] = \texttt{column\_compression}(L, D)$;
8: **end for**
9: $P = LDL^{\mathsf{T}}$.
10: $P = \texttt{gather}(P)$.

compression can be performed in various ways, usually by computing either a reduced rank-revealing QR factorization or a reduced SVD [30]. Here, we employ a reduced SVD factorization, followed by a diagonalization of the small resulting system. It is cheap as long as the rank of the solution stays low, which is the case in many applications.

As noted in Section 2, we also want to approximate the solutions to DREs and generalized DLEs and DREs. Therefore, we further have to solve the subproblems given by F_3 and F_4. Pseudo-code for these computations, based on the low-rank factorizations given in Section 2.2–2.3, is shown in Algorithms 3.3 – 3.4.

Algorithm 3.3 Solving $\dot{P} = F_3(P)$ on the GPU

1: Given: B, R^{-1}, h and a low-rank factorization of $P = LDL^{\mathsf{T}}$ on the GPU.
2: Compute $D = (I + hDL^{\mathsf{T}}BR^{-1}L)^{-1}D$;
3: $P = LDL^{\mathsf{T}}$.

Algorithm 3.4 Solving $\dot{P} = F_4(P)$ on the GPU

1: Given: S, h and a low-rank factorization of $P = LDL^{\mathsf{T}}$ on the GPU.
2: Compute $L = [L, \sqrt{h}SL, h/\sqrt{2}\,S^2L]$;
3: Compute $D = \texttt{blkdiag}(D, D, D)$;
4: $[L, D] = \texttt{column\_compression}(L, D)$;
5: $P = LDL^{\mathsf{T}}$.

We use three approaches to split the DRE: First, we apply Algorithm 3.2 to solve the Lyapunov part of the equation and compute $\mathcal{T}_3$ by Algorithm 3.3, forming

$$\mathcal{T}_{12}\left(\frac{h}{2}\right)\mathcal{T}_3\left(h\right)\mathcal{T}_{12}\left(\frac{h}{2}\right)P_0.$$

Further, we consider the three-term splitting

$$\mathcal{T}_1\left(\frac{h}{2}\right)\mathcal{T}_2\left(\frac{h}{2}\right)\mathcal{T}_3\left(h\right)\mathcal{T}_2\left(\frac{h}{2}\right)\mathcal{T}_1\left(\frac{h}{2}\right)P_0,$$

where we compute the two terms $\mathcal{T}_1(h)P_0$ and $\mathcal{T}_2(h)P_0$ as in Algorithm 3.1. Finally we reverse the order of the three-term splitting

$$\mathcal{T}_1\left(\frac{h}{2}\right)\mathcal{T}_3\left(\frac{h}{2}\right)\mathcal{T}_2\left(h\right)\mathcal{T}_3\left(\frac{h}{2}\right)\mathcal{T}_1\left(\frac{h}{2}\right)P_0.$$

Due to the additional splitting term, further errors are introduced, but since the integral does not have to be computed the three-term splitting codes are less computationally demanding.

The generalized DLE can be solved by the same three approaches. Using Algorithm 3.4, $\mathcal{T}_3$ is replaced by $\mathcal{T}_4$ in the previous three formulas. Finally, we apply a three-term Strang splitting to the generalized DRE, given by

$$\mathcal{T}_{12}\left(\frac{h}{2}\right)\mathcal{T}_3\left(\frac{h}{2}\right)\mathcal{T}_4\left(h\right)\mathcal{T}_3\left(\frac{h}{2}\right)\mathcal{T}_{12}\left(\frac{h}{2}\right)P_0$$

is applied.

4. Numerical experiments

The aim of this section is to show the different splitting strategies applied to the matrix equations, using the algorithms described in the previous section. We will first show the accuracy of the splitting schemes on a small-scale example, where we can compute an accurate reference solution by a Matlab built-in solver. Then, we show the speed-up of the code of the GPU implementation in comparison to the CPU implementation and compare the efficiencies of the various methods. Finally, we consider two real-world medium- to large-scale examples for DLE and DRE, respectively, and demonstrate that GPU acceleration is similarly advantageous there.

4.1. Small-scale accuracy verification.

We show the results of the algorithms on a small example. Consider the Laplacian on the unit square with homogeneous Dirichlet boundary conditions. By discretizing it using central second-order finite differences with n grid points in each space dimension, we acquire a matrix $A \in \mathbb{R}^{n^2\times n^2}$. We let Q and P_0 be randomly chosen matrices of rank 2 and rank 5, respectively. The tolerance for both the column-compression and the Leja interpolation were set to 10^{-16}.

Figure 1 shows an order plot for the Strang splitting applied to the DLE (1). The reference solution is computed by the Matlab routine `ode15s` with relative tolerance $2.22 \cdot 10^{-14}$ and absolute tolerance 10^{-20}. We take as final time $T = \frac{1}{2}$ and $n = 5$.

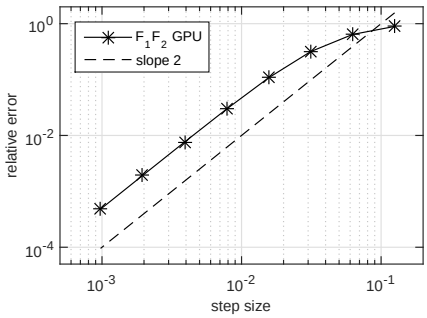

Figure 1. Relative error of the Strang splitting scheme applied to the differential Lyapunov equation.

Next, we apply the Strang splitting to the DRE with A, Q and P_0 as defined previously, B as a randomly chosen matrix of size $n^2 \times 1$ and either $R^{-1} = 1$ or

$R^{-1} = 10^{-3}$, see Figure 2. The DRE is solved by the three approaches introduced in the previous section. In the following, we will denote these by "2 term", "3 term $F_1F_2F_3$" and "3 term $F_1F_3F_2$". We see from Figure 2 that the three-term splitting

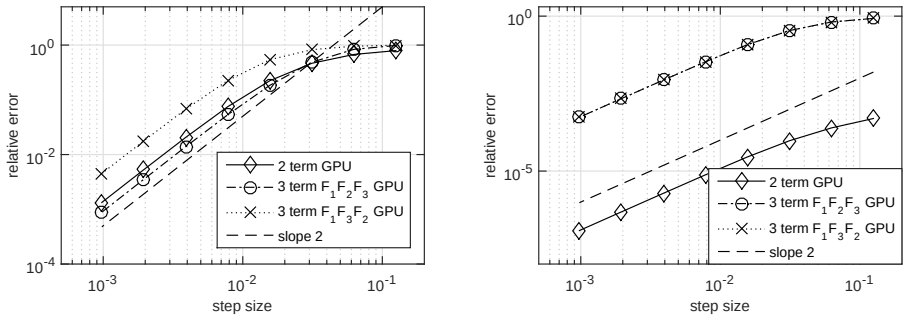

FIGURE 2. Relative error of the different splitting schemes applied to DRE with $R^{-1} = 1$ (left) and $R^{-1} = 10^{-3}$ (right).

$F_1F_3F_2$ is less accurate, whereas the errors of the two remaining splitting schemes behave similarly. Thus, the error due to splitting away the part F_3 is more severe than splitting F_1 and F_2. However, using $R^{-1} = 10^{-3}$ leads to a different result. The three-term splittings now yield roughly equally large errors, but the two-term splitting is about 10 times more accurate than the other schemes. Here we clearly observe the additional error introduced by the third splitting term.

Finally, we consider for the generalized matrix equations an example introduced in [13], where the matrix A denotes again the discretized $2D$ Laplacian on the unit square with homogeneous Dirichlet boundary conditions on two edges. On the third edge, we use the fixed boundary condition given by $x = u$, and on the final edge a Robin boundary condition $n\nabla x = 0.5(0.5 + dW)x$ is applied. This leads to a matrix $B \in \mathbb{R}^{n^2 \times 1}$ and a matrix $S \in \mathbb{R}^{n^2 \times n^2}$. The matrix $Q = CC^{\mathsf{T}}$ is defined by letting $C = \frac{1}{n^2}(1, \ldots, 1)$ be the matrix representation of the mean. We then solve the generalized Lyapunov equation (7) and show the corresponding error plot in Figure 3 (left). Moreover, we take $R = 1$ and compute also the relative error of the solution of the generalized Riccati equation, see Figure 3 (right).

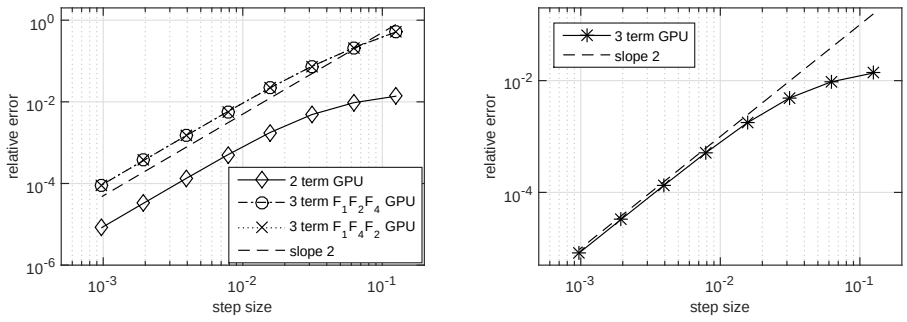

FIGURE 3. Relative errors of the different splitting schemes applied to the generalized DLE (left) and the generalized DRE (right).

We again see that the two-term splitting of the generalized DLE is approximately 10 times more accurate than the other two splitting schemes. As in all previous examples, we observe that the error of the generalized DRE behaves as expected, i.e. it converges with order 2 and remains small for all step sizes.

4.2. **GPU speed-up.** We compare the computational costs of the GPU based algorithm described above running on a Tesla K80 with 2×12 GB RAM, with the algorithm which operates only on the CPU (Intel Xeon E5-2630). We compute the solution until $T = \frac{1}{2}$ with step size $h = 0.005$ for different sizes of the matrix A and repeat the calculation 50 times to get an accurate result. We build the matrices as described in Subsection 4.1. All of the experiments are performed on Matlab on the same platform. In order to have a fair comparison between the different implementations, we run the Matlab codes on a single core by using the command `-singleCompThread`. Moreover, we deactivate the Java Virtual Machine by `-nojvm`. The computing time of the CPU algorithm is estimated by the command `tic - toc`. For the GPU algorithm we use the `wait` function with a `GPUDevice` object as input and measure the time by `tic - toc` and `wait`.

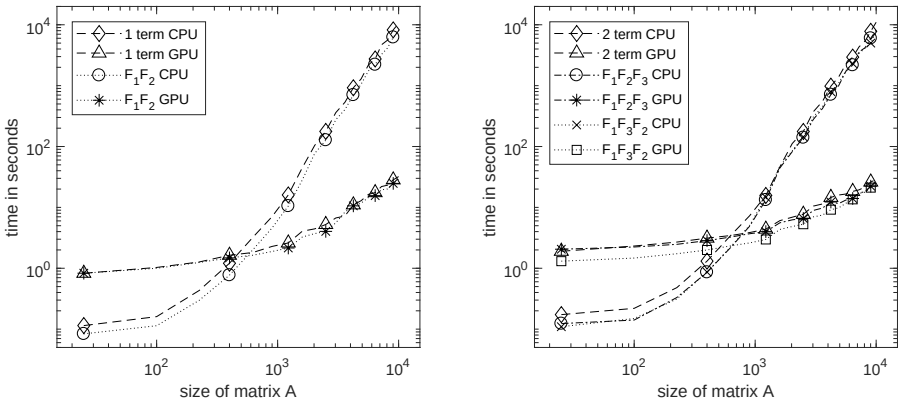

FIGURE 4. Computational costs of the algorithms for the DLE (left) and DRE (right) are given in a log-log plot for different sizes of the matrix A.

	$n = 5$	$n = 25$	$n = 50$	$n = 75$	$n = 100$
1 term CPU	$1.144 \cdot 10^{-1}$	$3.232 \cdot 10^{0}$	$1.769 \cdot 10^{2}$	$2.185 \cdot 10^{3}$	$1.057 \cdot 10^{4}$
1 term GPU	$8.280 \cdot 10^{-1}$	$1.817 \cdot 10^{0}$	$5.235 \cdot 10^{0}$	$1.634 \cdot 10^{1}$	$3.209 \cdot 10^{1}$
F_1F_2 CPU	$8.360 \cdot 10^{-2}$	$2.020 \cdot 10^{0}$	$1.274 \cdot 10^{2}$	$1.745 \cdot 10^{3}$	$8.135 \cdot 10^{3}$
F_1F_2 GPU	$8.298 \cdot 10^{-1}$	$1.638 \cdot 10^{0}$	$4.034 \cdot 10^{0}$	$1.453 \cdot 10^{1}$	$2.895 \cdot 10^{1}$

TABLE 1. Time measurements for DLE for different matrix sizes.

	$n = 5$	$n = 25$	$n = 50$	$n = 75$	$n = 100$
2 term CPU	$1.731 \cdot 10^{-1}$	$3.424 \cdot 10^{0}$	$1.766 \cdot 10^{2}$	$2.261 \cdot 10^{3}$	$1.100 \cdot 10^{4}$
2 term GPU	$1.902 \cdot 10^{0}$	$3.446 \cdot 10^{0}$	$7.720 \cdot 10^{0}$	$1.659 \cdot 10^{1}$	$2.875 \cdot 10^{1}$
$F_1F_2F_3$ CPU	$1.233 \cdot 10^{-1}$	$2.176 \cdot 10^{0}$	$1.401 \cdot 10^{2}$	$1.725 \cdot 10^{3}$	$8.174 \cdot 10^{3}$
$F_1F_2F_3$ GPU	$2.055 \cdot 10^{0}$	$3.252 \cdot 10^{0}$	$6.577 \cdot 10^{0}$	$1.321 \cdot 10^{1}$	$2.585 \cdot 10^{1}$
$F_1F_3F_2$ CPU	$1.096 \cdot 10^{-1}$	$2.227 \cdot 10^{0}$	$1.436 \cdot 10^{2}$	$1.773 \cdot 10^{3}$	$7.755 \cdot 10^{3}$
$F_1F_3F_2$ GPU	$1.308 \cdot 10^{0}$	$2.332 \cdot 10^{0}$	$5.293 \cdot 10^{0}$	$1.171 \cdot 10^{1}$	$2.509 \cdot 10^{1}$

TABLE 2. Time measurements for DRE for different matrix sizes.

In Figure 4 (left) the differential Lyapunov equation is solved with the splitting scheme as described in Section 2.1, see also Algorithm 3.1, and by the non-splitting

scheme, where we compute the integral by a high-order Gauss quadrature rule as in Algorithm 3.2. The computing time is given as a function of the size of the matrix A. We observe that for small matrices the CPU implementation is less time consuming than the GPU implementation. However, if the size of the problem exceeds $10^3 \times 10^3$, it pays off to use the GPU implementation. We observe a speed-up of over a factor 280 for matrices of size $10^4 \times 10^4$, for both splitting schemes. We further note that the time complexity of the splitting scheme is comparable to the non-splitting part. A similar behaviour can be seen in Figure 4 (right), where we plot the measured time of the splitting schemes of the DRE. Again, a major speed-up of the implementation on the GPU is detected for these medium-scale problems. (For matrices of size $10^4 \times 10^4$ the GPU implementation is approximately 300 times faster). We expect an even larger performance gain for truly large-scale problems.

We see no difference between the two three-term splitting schemes regarding time measurements, but the two-term splitting is slightly slower (as expected). The lack of speed is often more than compensated by a higher accuracy, however, as shown in the efficiency plot in Figure 5. This plots the relative error against the required computation times, and thus the further left and down in the plot, the "better". We observe that the three-term schemes are most efficient for all error levels when $R = 1$, while the two-term splitting is more efficient for small error levels when $R = 10^{-3}$. This makes sense, as the former case means that Q is relatively small and splitting it away therefore yields a small splitting error. In the latter case, Q is relatively big, and should not be split. (Because we are considering a very small-scale example here, the CPU versions of the code outperform the GPU in all cases.)

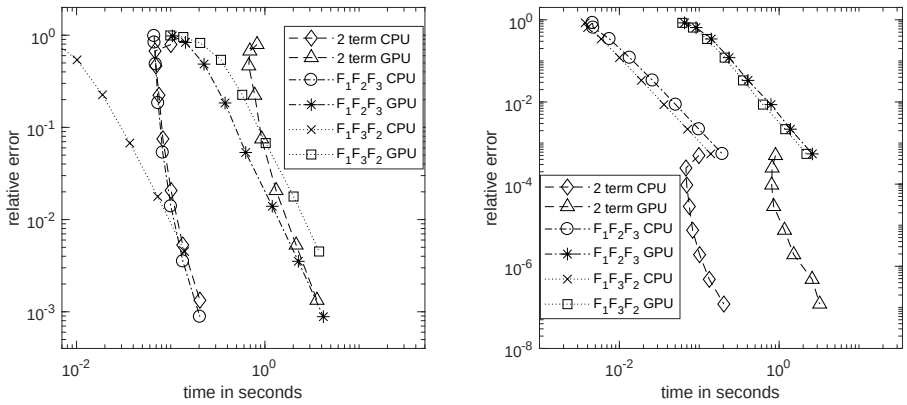

FIGURE 5. Efficiency of the splitting algorithms for the DRE for $n = 5$ and $R = 1$ (left) or $R^{-1} = 10^{-3}$ (right).

The GPU-based codes also exhibit better performance for the generalized matrix equations, see Figure 6. We consider here a two-term splitting of the generalized DLE as well as two different three-term splittings, and compare the computational costs of these schemes as well as the difference between CPU and GPU implementation. We use the matrices for the equations that were introduced in the previous subsection. As mentioned in Section 2, we employ a three-term splitting for the generalized DRE. We do not consider a four-term splitting here, since the error constants can be expected to be prohibitively large.

4.3. **Real-world example: Simulation of El Niño.** As a second example for DLEs, we consider the weather phenomenon El Niño. This is characterized by an

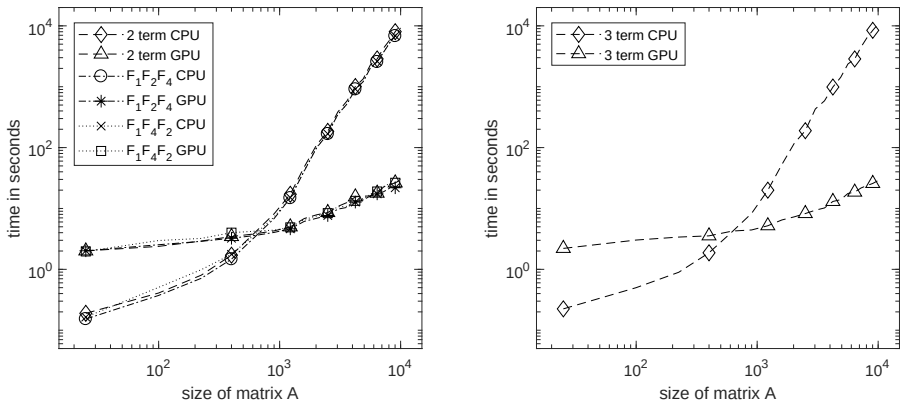

FIGURE 6. Computational costs of the algorithm for the generalized DLE (left) and the generalized DRE (right) are given in a log-log plot for different sizes of the matrix A.

	$n = 5$	$n = 25$	$n = 50$	$n = 75$	$n = 100$
2 term CPU	$1.893 \cdot 10^{-1}$	$4.028 \cdot 10^{0}$	$1.846 \cdot 10^{2}$	$2.283 \cdot 10^{3}$	$1.060 \cdot 10^{4}$
2 term GPU	$2.008 \cdot 10^{0}$	$3.826 \cdot 10^{0}$	$8.573 \cdot 10^{0}$	$1.604 \cdot 10^{1}$	$2.928 \cdot 10^{1}$
$F_1F_2F_4$ CPU	$1.527 \cdot 10^{-1}$	$3.411 \cdot 10^{0}$	$1.676 \cdot 10^{2}$	$2.029 \cdot 10^{3}$	$9.104 \cdot 10^{3}$
$F_1F_2F_4$ GPU	$2.018 \cdot 10^{0}$	$3.596 \cdot 10^{0}$	$7.669 \cdot 10^{0}$	$1.603 \cdot 10^{1}$	$2.262 \cdot 10^{1}$
$F_1F_4F_2$ CPU	$1.642 \cdot 10^{-1}$	$3.762 \cdot 10^{0}$	$1.679 \cdot 10^{2}$	$2.074 \cdot 10^{3}$	$9.239 \cdot 10^{3}$
$F_1F_4F_2$ GPU	$1.967 \cdot 10^{0}$	$4.229 \cdot 10^{0}$	$8.375 \cdot 10^{0}$	$1.691 \cdot 10^{1}$	$3.069 \cdot 10^{1}$

TABLE 3. Time measurements for DLE for different matrix sizes.

unusual warming of the sea surface temperature in the Indo-Pacific ocean. It can be modeled by a stochastic advection equation driven by additive noise [35] and its covariance is given by a DLE of the form

$$\dot{P}(t) = AP(t) + P(t)A^{\mathsf{T}} + Q,$$

see [31, 32] for details. The matrix A arises from a centered finite difference approximation of the advection operator and Q is the discretized covariance operator of the random noise. We consider here the different discretization resolutions corresponding to $n = 624, 1740, 3900, 7800$ and 15600, and approximate the solution either by Algorithm 3.1 or by directly approximating the integral term in (4) as outlined in section 2.1.

Figure 7 (left) shows the relative error for the splitting schemes for different step sizes. The reference solution is computed by the non-splitting scheme with a 16 times smaller step size. We show only the results for $n = 1740$, but the error behaves similarly for the other problem sizes. We observe the second-order convergence of the Strang splitting scheme and note that the relative error is small even for larger step sizes.

Similar to the academic example from the previous section, the time complexities of the non-splitting scheme and the splitting scheme are comparable, as shown in Figure 7 (right). The problem size where the GPU parallelization starts to pay off is also similar: at $n = 624$ the CPU and GPU costs are comparable but at $n = 1740$ we already observe a speed-up of a factor 6. For the finest resolution, the speed-up is roughly a factor 100.

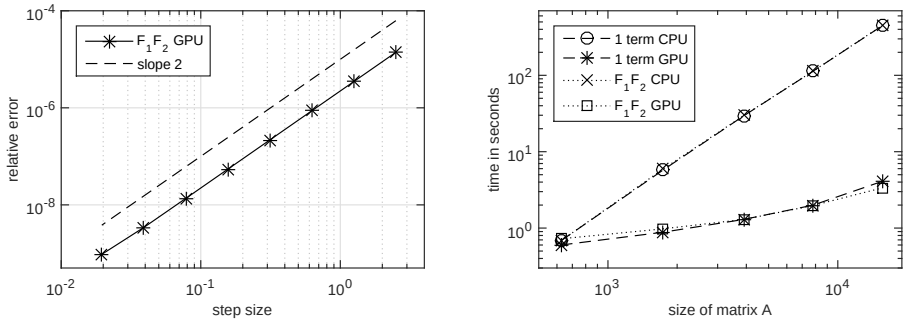

FIGURE 7. The relative error for $n = 1740$ for the splitting scheme of the DLE for the El Niño example (left) and the computational costs for different resolutions (right)

Figure 8 displays the efficiency of the numerical schemes for the El Niño example. As indicated by Figure 7 (right), the GPU and CPU versions of the 2-term splitting are roughly as efficient for the smallest problem. The 1-term "splitting" performs much better, and yields the same error independent of the step size. This happens because we are essentially computing the exact solution to the problem, and the only error is the quadrature error arising from the integral approximation. The same behaviour can also be observed for $n = 3900$, but here the GPU implementation clearly outperforms the CPU implementation. The 1-term method is again most efficient, but we note that the shape of the curves indicates that the 2-term splitting will become advantageous for either smaller errors or larger problems.

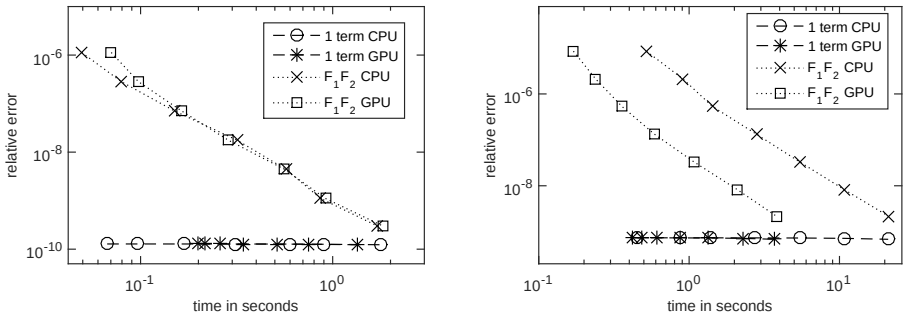

FIGURE 8. Efficiency of the splitting algorithms for the El Niño example for $n = 624$ (left) and $n = 3900$ (right).

4.4. **Real-world example: Steel cooling.** As a second example for DREs, we consider the optimal cooling of steel profiles. This problem has been widely studied in the literature, for details see [39, 16]. It gives rise to a DRE of the form

$$M^\mathsf{T}\dot{P}M = A^\mathsf{T}PM + M^\mathsf{T}PA + Q - M^\mathsf{T}PBR^{-1}B^\mathsf{T}PM.$$

The matrices M and A are the mass and stiffness matrices resulting from a finite element discretization of the Laplacian on a non-convex polygonal domain (the steel profile). Q is chosen as CC^T, where C is the discretization of an operator that measures temperature differences between different points in the domain. Finally, the matrix B is the discretization of the operator that implements the Neumann

boundary conditions of the Laplacian – this results in a boundary control application. Cancelling M^T and M leads to the equation

$$\dot{P} = M^{-\mathsf{T}}A^\mathsf{T}P + PAM^{-1} + M^{-\mathsf{T}}QM^{-1} - PBR^{-1}B^\mathsf{T}P,$$

which we can treat like outlined in Section 2.2 after replacing A by $M^{-1}A$ and Q by $M^{-\mathsf{T}}QM^{-1}$.

We note that we would normally never explicitly compute the (generally dense) matrices involving M^{-1}. In the CPU code, we form and reuse an incomplete LU decomposition of M to cheaply solve a linear equation system whenever the action of M^{-1} or $M^{-\mathsf{T}}$ is required. In the GPU code, the issue is unexpectedly complicated by the fact that Matlab's CUDA interface does not support solving equation systems with sparse system matrices and (dense) block right-hand sides. This *is* supported in the cuSPARSE library of CUDA itself, so until the Matlab interface is extended one might theoretically implement this capability by a MEX extension. In order to demonstrate performance gains by rather easy means, however, we do not do this. Instead, we compute and store a dense LU factorization. This is clearly not viable for truly large-scale problems, but problems of up to size $n \approx 3 \cdot 10^4$ are easily possible on our available hardware, and up to $n \approx 5.5 \cdot 10^4$ if AM^{-1} is explicitly formed at a slightly higher initial cost. Despite the heavy additional memory requirement, the GPU parallelization leads to a significant speed-up.

An additional issue related to the mass matrix is the original Leja point interpolation method for the computation of matrix exponential actions. One of the main steps of this algorithm computes an estimate of the spectrum of A by the use of Gershgorin discs. Since computing these require direct access to the elements in A, it is not directly applicable to AM^{-1} without explicitly forming the matrix. To get around this issue and still acquire a cheap estimate, we utilized the results of [33] which extends the Gershgorin approach to generalized eigenvalue problems. In our experience, this method overestimates the imaginary part of the spectrum but otherwise works well. We note that if the GPU code utilizes dense matrices, we may of course simply compute AM^{-1} and apply the original Leja point method. Since we expect to be able to work with sparse matrices in the near future, however, we follow the approach outlined above in both the CPU and GPU codes.

In the following we consider discretizations corresponding to $n = 371$, 1357, 5177 and 20209. We take $R^{-1} = I$, $T = \frac{1}{2}$ and $P(0) = 0$ and measure the computation times and relative errors of the different implementations. In Figure 9 (left) the relative error is shown for the size $n = 1357$ and different temporal step sizes. The reference solution is computed by the two-term splitting scheme with a step size that is 32 times smaller. We see that the two-term splitting performs considerably better than the other schemes, and the additional error due to a third splitting term has a huge impact on the approximation. In Figure 9 (right) we show the measured computation times for the algorithms when 100 time steps are taken. For the medium scale problem ($n = 1357$), the GPU and the CPU implementations have roughly the same computational costs but for the larger scale problems, the GPU implementation pays off.

We observe nearly no difference in the time measurements of the three different splitting schemes. This is likely because the integral term is only computed once, and the cost of this computation is minor in comparison to the remaining computations. Since the two-term splitting scheme is much more accurate here, it is therefore also more efficient, as shown in Figure 10. If only low accuracy is desired, the three-term splitting schemes are the best choice, but in all other cases the two-term splitting is most effective.

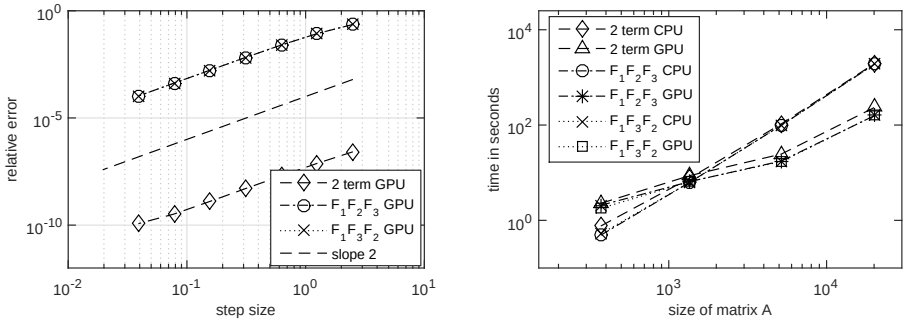

FIGURE 9. Left: Relative error of the steel example with $n = 1357$. Right: Computational costs of different splitting schemes applied to the steel example with $n = 371, 1357, 5177$ and 20209.

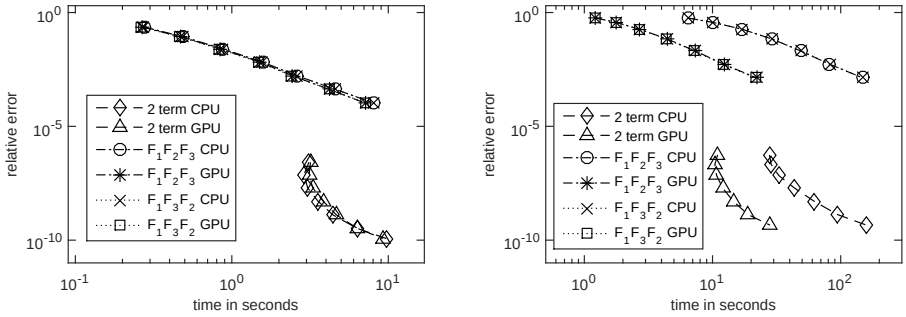

FIGURE 10. Efficiency of the splitting algorithms for the steel example with $n = 1357$ (left) and $n = 5177$ (right).

5. Conclusions

We have considered several different splitting schemes based on Leja point interpolation for the computation of matrix exponential actions. Since the matrix exponentials act on skinny block-matrices (the low-rank factors) rather than only vectors, we expected that these computations would be highly parallelizable and that GPU acceleration would therefore be beneficial. The latter was verified by several numerical experiments. In the considered problems of academical nature, the GPU code was faster than the pure CPU code by approximately a factor 10 already for matrices of size $2 \cdot 10^3$, and by a factor 10^3 for matrices of size 10^4. The break-even point was around size 650, which is well below what would be considered large-scale today. In the tested real-world applications, the gains in the El Niño example were in accordance with the more academic examples. For the steel example, they were more modest, but still around a factor 10 for the relevant problem sizes. As there is no difference in the size of the numerical errors, this clearly shows that GPU acceleration can lead to large gains in efficiency and should be considered for matrix equations of these type. The efficiency could additionally be further increased by considering more advanced parallelization techniques. An obvious such candidate is to investigate the use of single-precision computations when the desired level of accuracy is low.

We have also presented comparisons of different splitting strategies, mainly investigating whether one should split away the constant term Q or not, and in which order the subproblems should be solved. For the latter question, we observe that

the ordering has minimal influence on the error, and we may thus choose the order such that the computational cost is minimized. (E.g., take the most expensive subproblem as the "middle" term.) For the first question, we expected that it would not be beneficial to split away Q, since the extra integral term which arises only has to be approximated once. This was verified by our experiments, except in the case when Q was relatively small – then, of course, the extra splitting error is similarly small. We note that these results are for the autonomous case. When the matrices that define the equations also depend on time, the situation likely changes as the integral term would need to be recomputed in each step. However, as the modified methods would still rely on matrix exponential actions as their basic building blocks, we still expect that GPU acceleration would significantly increase the efficiency.

Acknowledgements

The authors are grateful to Peter Kandolf for his assistance with the original `expleja` code. H. Mena and L.-M. Pfurtscheller were supported by the Austrian Science Fund (FWF) - project id:P27926.

References

[1] H. Abou-Kandil, G. Freiling, V. Ionescu, and G. Jank, *Matrix Riccati equations in control and systems theory*, Birkhäuser, Basel, Switzerland, 2003.

[2] A. C. Antoulas, D. C. Sorensen, and Y. Zhou, *On the decay rate of Hankel singular values and related issues*, Syst. Cont. Lett. **46** (2002), no. 5, 323–342.

[3] N. Auer, L. Einkemmer, P. Kandolf, and A. Ostermann, *Magnus integrators on multicore CPUs and GPUs*, Comput. Phys. Comm. **228** (2018), 115–122.

[4] Winfried Auzinger, Othmar Koch, and Mechthild Thalhammer, *Defect-based local error estimators for high-order splitting methods involving three linear operators*, Numerical Algorithms **70** (2015), no. 1, 61–91.

[5] T. Başar and P. Bernhard, *H^∞-optimal control and related minimax design problems*, second ed., Systems & Control: Foundations & Applications, Birkhäuser Boston, Inc., Boston, MA, 1995, A dynamic game approach.

[6] J. Baker, M. Embree, and J. Sabino, *Fast singular value decay for Lyapunov solutions with nonnormal coefficients*, arXiv e-prints 1410.8741v1, Cornell University, October 2014, math.NA.

[7] Nathan Bell and Michael Garland, *Efficient sparse matrix-vector multiplication on CUDA*, NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[8] P. Benner and T. Breiten, *Low rank methods for a class of generalized Lyapunov equations and related issues*, Numerische Mathematik **124** (2013), no. 3, 441–470.

[9] P. Benner, E. Dufrechou, P. Ezzatti, H. Mena, E. S. Quintana-Ortí, and A. Remón, *Solving sparse differential Riccati equations on hybrid CPU-GPU platforms*, Computational Science and Its Applications – ICCSA 2017: 17th International Conference, Trieste, Italy, July 3-6, 2017, Proceedings, Part I (Osvaldo Gervasi, Beniamino Murgante, Sanjay Misra, Giuseppe Borruso, Carmelo M. Torre, Ana Maria A. C. Rocha, David Taniar, Bernady O. Apduhan, Elena Stankova, and Alfredo Cuzzocrea, eds.), Springer International Publishing, Cham, 2017, pp. 116–132.

[10] P. Benner, P. Ezzatti, H. Mena, E. S. Quintana-Ortí, and A. Remón, *Solving differential Riccati equations on multi-GPU platforms*, Proceedings of 11th International Conference on Computational and Mathematical Methods in Science and Engineering (Benidorm), CMMSE '11, 2011, pp. 178–188.

[11] P. Benner, P. Ezzatti, H. Mena, Quintana-Ortí E. S., and A. Remón, *Solving matrix equations on multi-core and many-core architectures*, Algorithms **6** (2013), no. 4, 857–870.

[12] P. Benner, Ezzatti P., Quintana-Ortí E. S., and A. Remón, *Unleashing CPU-GPU acceleration for control theory applications*, Euro-Par 2012: Parallel Processing Workshops - BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience, UCHPC, VHPC, Rhodes Islands, Greece, August 27-31, 2012. Revised Selected Papers (Ioannis Caragiannis, Michael Alexander, Rosa M. Badia, Mario Cannataro, Alexandru Costan, Marco Danelutto, Frédéric Desprez, Bettina Krammer, Julio Sahuquillo, Stephen L. Scott, and Josef Weidendorfer, eds.), Lecture Notes in Comput. Sci., vol. 7640, Springer, 2012, pp. 102–111.

[13] Peter Benner and Tobias Damm, *Lyapunov equations, energy functionals, and model order reduction of bilinear and stochastic systems*, SIAM Journal of Control and Optimization **49** (2011), no. 2, 686–711.
[14] Peter Benner and Hermann Mena, *Rosenbrock methods for solving Riccati differential equations*, IEEE Trans. Automat. Control **58** (2013), no. 11, 2950–2956.
[15] ———, *Numerical solution of the infinite-dimensional LQR problem and the associated Riccati differential equations*, J. Numer. Math. **26** (2018), no. 1, 1–20.
[16] Peter Benner and Jens Saak, *A semi-discretized heat transfer model for optimal cooling of steel profiles*, Dimension reduction of large-scale systems **45** (2005), 353–356.
[17] Marco Caliari, Peter Kandolf, Alexander Ostermann, and Stefan Rainer, *Comparison of software for computing the action of the matrix exponential*, BIT Numerical Mathematics **54** (2014), no. 1, 113–128.
[18] ———, *The Leja method revisited: Backward error analysis for the matrix exponential*, SIAM Journal on Scientific Computing **38** (2016), no. 3, A1639–A1661.
[19] Tobias Damm, Hermann Mena, and Tony Stillfjord, *Numerical solution of the finite horizon stochastic linear quadratic control problem*, Numerical Linear Algebra with Applications (2017).
[20] Mariano De Leo, Diego Rial, and Constanza Sánchez de la Vega, *High-order time-splitting methods for irreversible equations*, IMA Journal of Numerical Analysis **36** (2016), no. 4, 1842–1866.
[21] L. Einkemmer and A. Ostermann, *Exponential integrators on graphic processing units*, 2013 International Conference on High Performance Computing Simulation (HPCS), July 2013, pp. 490–496.
[22] Megan E. Farquhar, Timothy J. Moroney, Qianqian Yang, and Ian W. Turner, *GPU accelerated algorithms for computing matrix function vector products with applications to exponential integrators and fractional diffusion*, SIAM Journal on Scientific Computing **38** (2016), no. 3, C127–C149.
[23] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, *Understanding the performance of sparse matrix-vector multiplication*, 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), Feb 2008, pp. 283–292.
[24] Eskil Hansen and Alexander Ostermann, *High order splitting methods for analytic semigroups exist*, BIT. Numerical Mathematics **49** (2009), no. 3, 527–542.
[25] Eskil Hansen and Tony Stillfjord, *Convergence analysis for splitting of the abstract differential Riccati equation*, SIAM Journal on Numerical Analysis **52** (2014), no. 6, 3128–3139.
[26] Willem Hundsdorfer and Jan Verwer, *Numerical solution of time-dependent advection-diffusion-reaction equations*, Springer Series in Computational Mathematics, vol. 33, Springer-Verlag, Berlin, 2003.
[27] A. Ichikawa and H. Katayama, *Remarks on the time-varying H_∞ Riccati equations*, Syst. Cont. Lett. **37** (1999), no. 5, 335–345.
[28] Antti Koskela and Hermann Mena, *Analysis of Krylov Subspace Approximation to Large Scale Differential Riccati Equations*, ArXiv e-prints (2017).
[29] N. Lang, *Numerical methods for large-scale linear time-varying control systems and related differential matrix equations*, Dissertation, Technische Universität Chemnitz, Chemnitz, Germany, June 2017.
[30] N. Lang, H. Mena, and J. Saak, *On the benefits of the LDL^T factorization for large-scale differential matrix equation solvers*, Linear Algebra Appl. **480** (2015), 44–71.
[31] H. Mena, A. Ostermann, L.-M. Pfurtscheller, and C. Piazzola, *Numerical low-rank approximation of matrix differential equations*, J. Comput. Appl. Math. (2018), Accepted for publication.
[32] H. Mena and L. Pfurtscheller, *An efficient SPDE approach for El Niño*, ArXiv e-prints (2017).
[33] Yuji Nakatsukasa, *Gerschgorin's theorem for generalized eigenvalue problems in the Euclidean metric*, Mathematics of Computation **80** (2011), no. 276, 2127–2142.
[34] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron, *Scalable parallel programming with cuda*, Queue **6** (2008), no. 2, 40–53.
[35] C. Penland and P.D. Sardeshmukh, *The optimal growth of tropical sea surface temperature anomalies*, J. Clim. **8** (1995), 1999–2024.
[36] T. Penzl, *Eigenvalue decay bounds for solutions of Lyapunov equations: the symmetric case*, Syst. Cont. Lett. **40** (2000), 139–144.
[37] I. R. Petersen, V. A. Ugrinovskii, and A. V. Savkin, *Robust control design using h^∞ methods*, Springer-Verlag, London, UK, 2000.
[38] Jill Reese and Sarah Zaranek, *GPU programming in Matlab*, MathWorks News&Notes. Natick, MA: The MathWorks Inc (2012), 22–5.

[39] Jens Saak, *Effiziente numerische Lösung eines Optimalsteuerungsproblems für die Abkühlung von Stahlprofilen*, Ph.D. thesis, Diplomarbeit, Fachbereich 3/Mathematik und Informatik, Universität Bremen, D-28334 Bremen, 2003.

[40] D. C. Sorensen and Y. Zhou, *Bounds on eigenvalue decay rates and sensitivity of solutions to Lyapunov equations*, Tech. Report TR02-07, Dept. of Comp. Appl. Math., Rice University, Houston, TX, June 2002, Available online from http://www.caam.rice.edu/caam/trs/tr02.html#TR02-07.

[41] Tony Stillfjord, *Low-rank second-order splitting of large-scale differential Riccati equations*, IEEE Trans. Automat. Control **60** (2015), no. 10, 2791–2796.

[42] ———, *Adaptive high-order splitting schemes for large-scale differential Riccati equations*, Numerical Algorithms (2017).

Universidad Yachay Tech, Hacienda San José s/n, San Miguel de Urcuquí, Ecuador
E-mail address: mena@yachaytech.edu.ec

Universität Innsbruck, Technikerstrasse 13, A-6020 Innsbruck, Austria
E-mail address: Lena-Maria.Pfurtscheller@uibk.ac.at

Max Planck Institute for Dynamics of Complex Technical Systems, Sandtorstr. 1, DE-39106 Magdeburg, Germany
E-mail address: stillfjord@mpi-magdeburg.mpg.de