Numerical Algorithms
https://doi.org/10.1007/s11075-019-00687-w

ORIGINAL PAPER

GPU acceleration of splitting schemes applied to differential matrix equations

Hermann Mena[1,2] · **Lena-Maria Pfurtscheller**[2] · **Tony Stillfjord**[3]

Received: 22 May 2018 / Accepted: 7 March 2019 / Published online: 05 April 2019

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, large speed-ups 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.

Keywords Differential Lyapunov equations · Differential Riccati equations · Large scale · Splitting schemes · GPU acceleration

✉ Tony Stillfjord
stillfjord@mpi-magdeburg.mpg.de

Hermann Mena
mena@yachaytech.edu.ec

Lena-Maria Pfurtscheller
Lena-Maria.Pfurtscheller@uibk.ac.at

[1] Universidad Yachay Tech, Hacienda San José s/n, San Miguel de Urcuquí, Ecuador

[2] Universität Innsbruck, Technikerstraße 13, A-6020, Innsbruck, Austria

[3] Max Planck Institute for Dynamics of Complex Technical Systems, Sandtorstr. 1, DE-39106, Magdeburg, Germany

1 Introduction

We are interested in differential matrix equations of Lyapunov or Riccati type, or generalized versions of these. They are all of the following 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, G is a matrix-valued function of the solution $P \in \mathbb{R}^{n\times n}$. 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, and differential games [1, 6, 30, 40].

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 following 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 $Q = C^\mathsf{T}C$, 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.

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, 33] are usually outperformed by more modern methods such as Krylov-based projection schemes [31], peer methods [32], or splitting schemes [34, 44, 45]. 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 for 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 [11–13] as well as for their steady-state versions: the algebraic Lyapunov and Riccati equations [13, 16]. 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 [4, 22] (see also [25]). 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., [8, 22, 26] less relevant.

Since the relevant methods are mainly implemented in MATLAB, we restrict ourselves to utilizing its built-in GPU support [41] via NVIDIA's CUDA [37] 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 optimizations 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. For example, 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 in Section 3. The simple changes necessary for GPU utilization, and a discussion on what efficiency improvements may be expected is given in Section 4. The actual speed-ups are presented in Section 5, in the form of several numerical experiments. Finally, we summarize our conclusions in Section 6.

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 $\dot{x} = \Delta x + f(x)$ with homogeneous Dirichlet boundary conditions. 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. (For some caveats in the case of other boundary condition types, see e.g. [2, 23, 24].) 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 following 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 the following:

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

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., [29]). For simplicity, we restrict ourselves to the Strang splitting scheme in this paper, but one might also consider higher-order schemes [21, 27, 45], or schemes where the subproblems are not solved exactly (see, e.g., [28, 29]).

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 following 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 [5] 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., [3, 7, 9, 39, 43]); 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 following 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 the following:

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

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 follows:

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

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

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

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

where the integral term may be approximated by high-order quadrature as in [44]. 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 [44, 45] 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; thus, the larger error might be compensated for 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 the following:

$$\mathcal{T}_3(h)P_0 = (I + hP_0BR^{-1}B^\mathsf{T})^{-1}P_0. \tag{6}$$

A low-rank factorization is given by the following:

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

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

2.3 Generalized Lyapunov equations

We further consider a generalized Lyapunov equation of the following 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] the following:

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

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 [20]:

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

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.

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

2.4 Generalized Riccati equations

Moreover, we study a generalized Riccati equation given by the following:

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

and split this equation into three subproblems of the following 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 experience suggests that the extra error due to the splitting would become prohibitively large.

3 Implementations

In this section, we describe the implementation of the Strang splitting scheme applied to the differential matrix equations discussed in Section 2. Other splitting schemes such as the Lie splitting are implemented analogously.

In all the considered equations, the most demanding part is to compute the action of the matrix exponential in (2) efficiently. In [18, 19], 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 therefore use this method to compute $\mathrm{e}^{hA}L$ for different skinny matrices L, and denote it by `expleja` in the following.

First, we consider the DLE case. The discussion in Section 2.1 immediately leads to Algorithm 1.

Algorithm 1 Solving DLE by Strang splitting.

Given: A, Q, P_0, T, N_t, $h = \frac{T}{N_t}$.
Compute LDL^T-decompositions of $Q = L_Q D_Q L_Q^\mathsf{T}$ and $P_0 = LDL^\mathsf{T}$.
Compute parameters `param` for Leja interpolation.
for $k = 1, \ldots, N_t$ **do**
    $L = \texttt{expleja}(h/2, A, L, \texttt{param})$
    $L = [L, L_Q]$
    $D = \texttt{blkdiag}(D, hD_Q)$;
    $[L, D] = \texttt{column\_compression}(L, D)$;
    $L = \texttt{expleja}(h/2, A, L, \texttt{param})$
end for
$P = LDL^\mathsf{T}$;

On the other hand, as mentioned in Section 2.1, it is possible to derive an explicit form of the solution of the DLE given by (4). Following [45], we use a high-order quadrature rule to compute an approximation to the integral term. This computation is again based on using `expleja`, now to compute $\mathrm{e}^{s_k A}L_Q$ for various $s_k \in [0, h]$

with the LDL^T-factorization $Q = L_Q D_Q L_Q^\mathsf{T}$. This leads to the alternative Algorithm 2, which (as noted in Section 2.1) is not a splitting scheme per se.

Algorithm 2 Solving DLE by quadrature

1: Given: A, Q, P_0, T, N_t, $h = \frac{T}{N_t}$.
2: Repeat Steps 2 and 3 from Algorithm 1.
3: Approximate integral:

- Compute n nodes s_k and weights w_k of a 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: $\quad L = [\texttt{expleja}(h, A, L, \texttt{param}), L_I]$
6: $\quad D = \texttt{blkdiag}(D, D_I)$;
7: $\quad [L, D] = \texttt{column\_compression}(L, D)$;
8: **end for**
9: $P = LDL^\mathsf{T}$.

We note that in both Algorithm 1 and Algorithm 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 1 (for example) would add the columns in L_Q to L, while the rank would likely stay similar. The compression can be performed in various ways, usually by computing either a reduced rank-revealing QR factorization or a reduced SVD [33]. 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-codes for these computations, based on the low-rank factorizations given in Sections 2.2–2.3, are shown in Algorithms 3–4.

Algorithm 3 Solving $\dot{P} = F_3(P)$ over $[0, h]$.

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

Algorithm 4 Solving $\dot{P} = F_4(P)$ over $[0, h]$.

1: Given: S, h and a low-rank factorization of $P = LDL^\mathsf{T}$.
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 incorporate Algorithm 3 in Algorithm 2 in order to solve the Lyapunov part of the equation via quadrature and the nonlinear term via the exact solution formula in (6), forming the following:

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

Further, we consider the three-term splitting

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

by extending Algorithm 1 with a third step given by Algorithm 3. Finally, we reverse the order of the three-term splitting as follows:

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

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

The generalized DLE can be solved by the same three approaches. Using Algorithm 4, $\mathcal{T}_3$ is replaced by $\mathcal{T}_4$ in the previous three formulas. Finally, we consider a three-term Strang splitting for the generalized DRE, given by the following:

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

The modifications to Algorithms 1 and 2 for the (generalized) DRE and generalized DLE cases through use of Algorithms 3 and 4 are obvious, and we therefore omit full listings of these versions.

4 GPU considerations

All the algorithms in the previous section were implemented in MATLAB. For GPU acceleration, we used the Parallel Computing Toolbox, which interfaces with the CUDA library. This is a framework for general purpose computing on GPUs. Recent releases of MATLAB expose a large fraction of this framework as overloaded built-in functions, i.e., precompiled code that operates either on the CPU or the GPU, depending on where the data currently resides. Thus, e.g., solving a system of linear equations $Ax = b$ on the GPU can be accomplished by the familiar syntax `x = A\b` after A and b have been instantiated as objects on the GPU. This data transfer is performed by the `gpuArray` function. The result x may then be transferred back to the CPU by use of the `gather` function. We refer to, e.g., [41]. In general,

communication between the CPU and the GPU is expensive. We therefore first move all the data to the GPU, do all vector- and matrix-computations on the GPU using built-in functions and transfer only scalar quantities and the final results back to the CPU.

The main computational effort in all the algorithms is the computation of the matrix exponential actions via the `expleja` code. This consists of a (one-time) estimation of the spectrum of A by the Gershgorin disk theorem, a (one-time) computation of exponential interpolation parameters, and a Newton interpolation [18, 19]. These functions all depend only on matrix-vector or matrix-matrix products and simple built-in functions like `diag`, `sum`, and `abs`, all of which have overloaded GPU versions. There is thus no need for any changes to the main code.

4.1 Main routines and limiting factors

As will be demonstrated in Section 5, around 90–95% of the total computation time is spent in the Newton interpolation part of the `expleja` code. On the CPU side, this can be further broken down into the multiplication of a sparse matrix with a dense skinny matrix (65–75%), the computation of the 1-norm of a dense skinny matrix (15–25%), and the addition of two skinny matrices (5%). On the GPU side, the ranking of these operations are typically the same, but the relative percentages differ.

All of these operations are memory-bound, i.e., their computation is limited by memory bandwidth rather than processing power. This can easily be confirmed by considering the number of necessary read/writes compared to the number of actual floating point operations.

4.2 Possible perfomance gains

Since modern GPUs feature larger memory bandwidths than comparable CPUs and since the main operations are also highly parallelizable, we expect to see a speed-up when utilizing the GPU. This speed-up will likely not be as large as for a compute-bound problem, where GPUs excel, but should still be significant, especially considering the essentially zero cost of extra implementation effort.

If both the GPU and CPU operated at peak performance, the observed speed-up would simply be the ratio of the respective memory bandwidths. This will, however, not be the case in practice. Still, one might expect that both platforms operate at a similar percentage of peak performance, and that the ratio will stay similar. In practice, however, this will also not be the case, due to differences in code optimization. In the current application, the overwhelming majority of the computations are performed in very low-level operations which we can not influence. Since MATLAB is not open source, we have no insight into what particular algorithm is used or how well it is optimized. Because the main operations are memory-bound, efficient memory allocation also plays a large role. Here, again, we have no insight into what strategies MATLAB follows. Finally, the CPU typically also has one additional layer of (larger) cache memory than the GPU, which further complicates things. For these reasons, it is difficult to predict what kind of speed-up to expect.

5 Numerical experiments

The aim of this section is to apply the different splitting strategies to various examples and demonstrate that the GPU implementation consistently outperforms the CPU version, often by a large margin.

We first describe a number of examples, including two arising from real-world problems. As an implementation verification, we then test our codes on the first small-scale problem, where we can compute an accurate reference solution by vectorization of the problem. We observe the correct orders of convergence and also verify that the GPU and CPU codes indeed give the same results. Then, we compare the speed of the two platforms by applying the different algorithms to the given test examples, and demonstrate that GPU acceleration is advantageous in all cases.

The tests were run on two different systems. The first one, hereafter referred to as "System 1", has an Intel Xeon E5-2630v3 CPU and a Tesla K80. The K80 contains two separate GPUs, of which we use only one. The maximum memory bandwidths are here 59 GB/s[3] for the CPU and 240 GB/s[4] for the GPU. This system has 24 GB RAM available. The second system, hereafter referred to as "System 2", is one node of the Mechthild[5] HPC cluster at the Max Planck Institute Magdeburg. This has an Intel Xeon Silver 4110 CPU and a Tesla P100 GPU. The maximum memory bandwidths are here 115 GB/s[6] for the CPU and 732 GB/s[7] for the GPU. This system has 192 GB RAM available.

In all our experiments, we use the tolerance 10^{-16} for both the column compression and the Leja interpolation. This ensures that the approximations are not unnecessarily truncated, and that the matrix exponential actions are essentially exact. We use MATLAB R2017a on System 1 and R2017b on System 2. In both cases, we deactivate the Java Virtual Machine by `-nojvm`. The computing times of the CPU algorithms are estimated by the command `tic - toc`. For the GPU algorithms, we do the same, except that we also call the `wait` function to ensure that all threads on the GPU have finished their computations before the measurements.

5.1 Experiment descriptions

5.1.1 Example 1: Heat equation random model

We first consider the Laplacian on the unit square with homogeneous Dirichlet boundary conditions. By discretizing it using central second-order finite differences with n_x grid points in each space dimension, we acquire a matrix $A \in \mathbb{R}^{n\times n}$ with $n = n_x^2$. We let Q and P_0 be randomly chosen matrices of rank 2 and rank 5, respectively and take B to be a randomly chosen matrix of size $n \times 1$. This corresponds

[3] https://ark.intel.com/products/83356/Intel-Xeon-Processor-E5-2630-v3-20M-Cache-2_40-GHz
[4] https://www.nvidia.com/en-us/data-center/tesla-k80/
[5] http://www.mpi-magdeburg.mpg.de/cluster/mechthild
[6] https://ark.intel.com/products/123547/Intel-Xeon-Silver-4110-Processor-11M-Cache-2_10-GHz
[7] https://www.nvidia.com/en-us/data-center/tesla-p100/

to optimal (distributed) control of the heat equation, with a single input and two outputs, and gives rise to a DRE. By ignoring the B matrix, we get instead a DLE where the solution corresponds to the time-limited Gramian of the system. In both cases, we use the final time $T = \frac{1}{2}$.

5.1.2 Example 2: Stochastic heat transfer

For the generalized matrix equations, we consider an example introduced in [10] arising from a stochastic heat transfer problem. The matrix A again denotes the discretized $2D$ Laplacian on the unit square, but now with homogeneous Dirichlet boundary conditions on two edges. On the third edge, we implement control through the fixed boundary condition $x = u$, and on the final edge a stochastic Robin boundary condition $n \cdot \nabla x = 0.5(0.5 + dW)x$ is applied, where W is a Brownian motion. This leads to a matrix $B \in \mathbb{R}^{n\times 1}$ and a (sparse) matrix $S \in \mathbb{R}^{n\times n}$. The matrix $Q = CC^{\mathsf{T}}$ is defined by letting $C = \frac{1}{n}(1, \ldots, 1)$ be the matrix representation of the mean. Similarly to the previous example, we may acquire a generalized DLE instead by simply ignoring the matrix B. (Then, there is a homogeneous Dirichlet boundary condition also on the third edge.) In both cases, we use the final time $T = \frac{1}{2}$.

5.1.3 Example 3: Simulation of El Niño

As a third example, we consider the real-world weather phenomenon El Niño. This is characterized by an 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 [38] and its covariance is given by a DLE of the following form:

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

see [34, 35] 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 $n = 624$, 3900, 7800, and 15600 and use the final time $T = 100$. We note that this problem only yields to a DLE.

5.1.4 Example 4: Simulation of steel cooling

For our final example, we consider the optimal cooling of steel profiles. This problem has been widely studied in the literature, for details, see [17, 42]. It gives rise to a DRE of the following 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 $C^{\mathsf{T}}C$, where C is the discretization of an operator that measures temperature differences between different points in the domain. (We want an even temperature distribution.) 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. Canceling M^T and M leads to the following 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 as 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 will lead 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 requires 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 [36] 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 the discretizations corresponding to $n = 371$, 1357, 5177, and 20209, for which the matrices have been precomputed . We take $R^{-1} = I$, $P(0) = 0$, and integrate until $T = 450$.

5.2 Implementation verification

In order to verify our implementations, we investigate the convergence properties of the methods when applied to Example 1 in Section 5.1.1 and Example 2 in Section 5.1.2 with $n = 25$. The reference solutions are computed by vectorizing the system and applying the MATLAB routine `ode15s` with relative tolerance $2.22 \cdot 10^{-14}$ (which is the lowest possible relative tolerance) and absolute tolerance 10^{-20}. We show only the results from the GPU versions of the code on System 1 to minimize clutter, but the CPU versions yield the same results and so do the simulations on System 2.

The left plot of Fig. 1 shows an order plot for the Strang splitting (Algorithm 1) applied to the DLE (1) arising from Example 1 in Section 5.1.1, and the right plot shows the corresponding results for the quadrature rule method (Algorithm 2). Here, and in the following, we identify the methods in the figure legends by in which order the subproblems are solved. Thus, the splitting in this example is written as $F_1 F_2$ and the quadrature scheme is denoted by F_{12}.

We note that the Strang splitting achieves second-order convergence as expected. The quadrature rule, on the other hand, yields a constant but very low error. This is in fact also the expected result, and the error is the error of the quadrature approximation to the integral. While Strang splitting has recently been suggested multiple times for DLEs, the extra cost for the quadrature rule is in our implementation only 14 additional evaluations of the matrix exponential action; we therefore expect that the quadrature rule will essentially always outperform the splitting. This is confirmed in the next section.

The situation is different for DREs, where we can split into either two or three terms, and the results depend on how large the nonlinear term is compared to the constant term. We consider the three approaches for DREs outlined in Section 3 and denote them by $F_{12}F_3$ (quadrature for the DLE part), $F_1F_2F_3$ and $F_1F_3F_2$. In Fig. 2, we show an order plot for these methods applied to the DRE (5) arising from Example 1 in Section 5.1.1. The left plot uses $R^{-1} = 1$ and the right one $R^{-1} = 10^{-3}$.

The first observation to be made is that all the methods converge with the correct order. We also see that the three-term splitting $F_1F_3F_2$ is less accurate when $R^{-1} = 1$, whereas the errors of the two remaining splitting schemes behave similarly. Thus, the error due to splitting off the part F_3 is more severe than splitting F_1 and F_2. This is because the nonlinear term is the dominant part here. Using instead $R^{-1} = 10^{-3}$ means that it is less significant, and leads to a different result. We see that the three-term splittings now yield roughly equally large errors, but that the two-term splitting is about ten times more accurate than the other schemes. Here, we clearly observe the additional error introduced by the third splitting term.

We also solve the generalized DLE (7) arising from Example 2 in Section 5.1.2 by the three methods mentioned in Section 3 and show the corresponding errors in Fig. 3

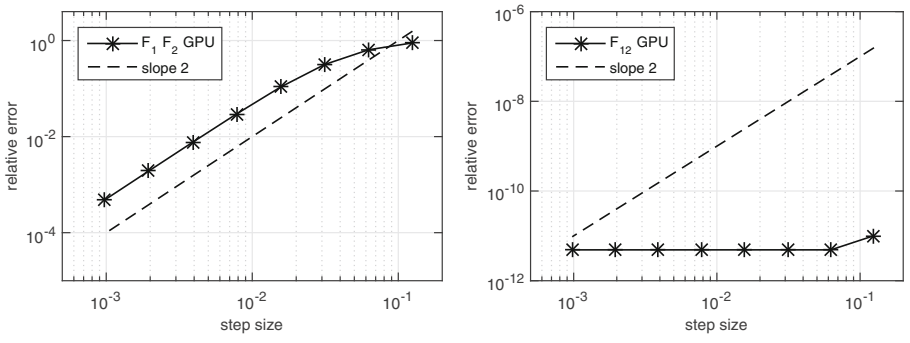

Fig. 1 Relative errors of the Strang splitting scheme (left) and the quadrature rule method (right) applied to the DLE in Example 1 in Section 5.1.1

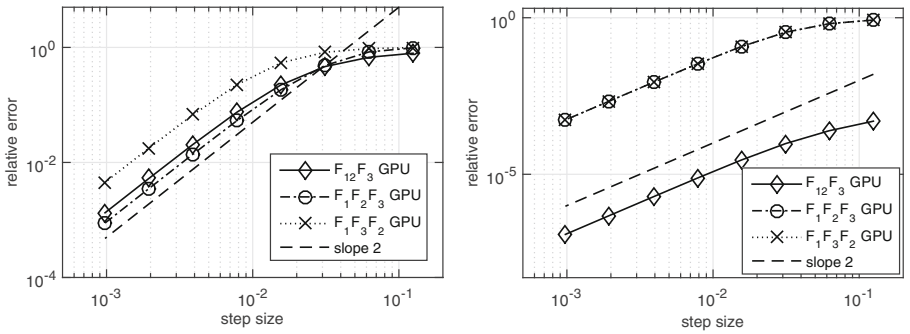

Fig. 2 Relative error of the different splitting schemes applied to DRE with $R^{-1} = 1$ (left) and $R^{-1} = 10^{-3}$ (right).

(left). Moreover, we take $R = 1$ and solve also the generalized DRE with the three-term Strang splitting and present the resulting errors in Fig. 3 (right) We again see that the two-term splitting of the generalized DLE is approximately ten times more accurate than the other two splitting schemes. As in all previous examples, we also observe that the error of the generalized DRE behaves as expected, i.e., it converges with order two and remains small for all step sizes.

Finally, we test our implementation also on the larger Example 3 in Section 5.1.3 and Example 4 in Section 5.1.4. Unlike the previous small-scale examples, we do not have a similarly exact reference solution. We instead compare our approximations to an approximation computed with the same scheme, but with a 16 times smaller step size. The left plot of Fig. 4 shows the relative error of the quadrature scheme versus the step sizes when applied to the DLE arising in Example 3 in Section 5.1.3, and the right plot shows the errors of the two-term and two different three-term splitting schemes applied to the DRE arising from Example 4 in Section 5.1.4. The problem sizes are here $n = 1740$ and $n = 1357$, respectively, but the error behaves similarly for the other problem sizes. We see that the quadrature rule again produces an essentially exact solution regardless of step size, while the splitting schemes all converge

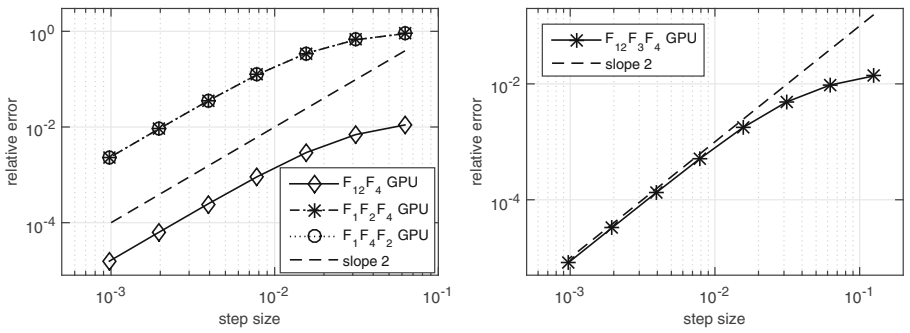

Fig. 3 Relative errors of the different splitting schemes applied to the generalized DLE (left) and the generalized DRE (right)

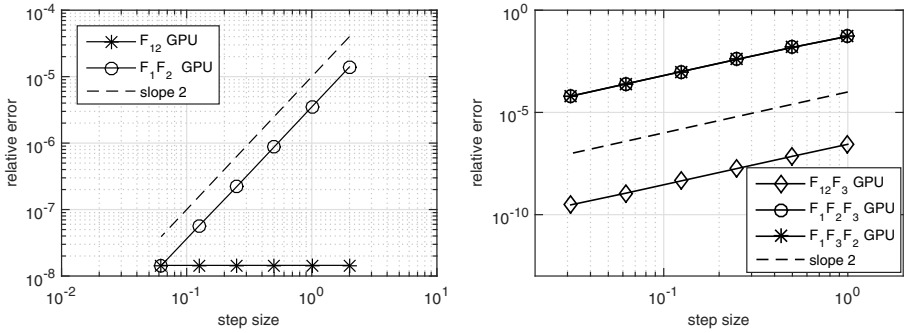

Fig. 4 The relative error for the quadrature rule applied to the El Niño DLE with $n = 1740$ (left) and the relative error for three different splitting schemes applied to the steel cooling DRE with $n = 1357$ (right)

with order two. The two-term splitting once again yields a much lower error than the three-term versions.

5.3 Performance on main sub-functions

In the following, we show the performance of the main sub-functions for the quadrature rule applied to the DLE arising from Example 1 in Section 5.1.1 with $n = 22500$. The other methods and problem cases behave similarly. Using the MATLAB profiler, one can show that in the CPU implementation around 98% of the total time is spent in the `explejа`, which in turn spends almost all its time in the Newton interpolation function. For the GPU implementation, the corresponding value is about 90% of the total cost. Hence, we focus on the main sub-functions in this Newton algorithm. They consist of the multiplication of a sparse matrix with a dense skinny matrix (denoted SpMM in the following), the computation of the 1-norm of a dense skinny matrix (1-norm) and the addition of two skinny matrices (addition). The time spent in these main sub-functions, relative to the total computation time, is shown in Table 1 for the CPU and GPU versions of the code and both systems.

As already mentioned in Section 4.2, we do not know which algorithm MATLAB uses for these sub-functions, but we can compare their costs on the CPU and GPU. We show in Fig. 5 the computational costs for the three main sub-functions, applied to randomly generated skinny matrices of ranks 15, 30, and 45. These ranks correspond

Table 1 Relative costs of the main sub-functions, in terms of the total computation time

Machine		Newton	SpMM	1-norm	addition
System 1	CPU	98.6%	72.4%	19.3%	4.4%
	GPU	93.3%	40.4%	36.2%	15.4%
System 2	CPU	98.3%	65.2%	27.1%	3.8%
	GPU	89.4%	35.8%	36.9%	14.0%

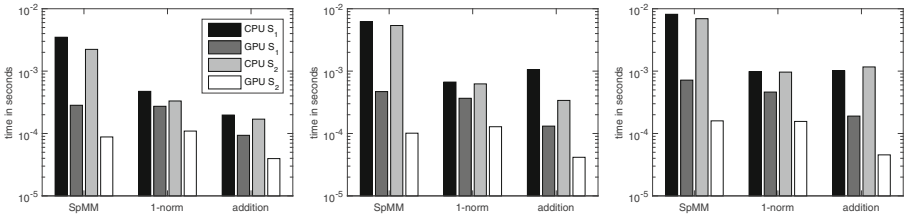

Fig. 5 Logarithmic plot of the computational costs of the three main operations for matrices with rank 15 (left), rank 30 (middle), and rank 45 (right)

to the typical ranks of the solutions to the matrix equations arising from Example 1 in Section 5.1.1 and Example 2 in Section 5.1.2. Here, and in the following, we denote the different systems by S_1 and S_2 in the figure legends, to save space.

We see that for System 1, we obtain a GPU speed-up of a factor 11–13 depending on the rank. This is higher than the expected theoretical factor 4.07. For the computation of the norm, however, we get only a factor of 1.8. The speed-up of the addition varies between 2.1 and 8.0. These different numbers reflect different optimization strategies and uses of cache memory on the different platforms. Since the SpMM multiplication takes up more than 70% of the total costs for the CPU, we draw the conclusion that we can expect a speed-up which is higher than 4.

For System 2, the SpMM speed-up depends highly on the rank; we get a factor 25.3 for rank 15, 53.2 for rank 30, and 43.4 for rank 45. This is again higher than the theoretical factor 6.37. The speed-up of the 1-norm is between 3.0 and 6.2 and the addition varies between 4.2 and 25.7. We thus again expect to see a speed-up higher than what would be expected if both the underlying libraries operated at peak efficicency.

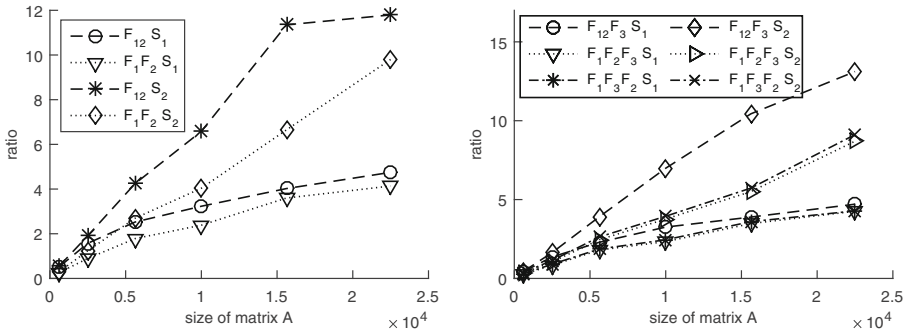

Fig. 6 Relative computational costs of the algorithms applied to the DLE (left) and DRE (right) arising from Example 1 in Section 5.1.1, for different problem sizes

Table 2 Computational costs of the algorithms applied to the DLE arising from Example 1 in Section 5.1.1, for different problem sizes (Wall-clock time, seconds)

	n	2500	5625	10000	15625	22500
System 1	F_{12} CPU	$8.3 \cdot 10^0$	$3.5 \cdot 10^1$	$8.3 \cdot 10^1$	$1.9 \cdot 10^2$	$4.1 \cdot 10^2$
	F_{12} GPU	$5.3 \cdot 10^0$	$1.4 \cdot 10^1$	$2.6 \cdot 10^1$	$4.8 \cdot 10^1$	$8.6 \cdot 10^1$
	F_1F_2 CPU	$4.4 \cdot 10^0$	$2.0 \cdot 10^1$	$4.6 \cdot 10^1$	$1.0 \cdot 10^2$	$2.0 \cdot 10^2$
	F_1F_2 GPU	$4.9 \cdot 10^0$	$1.1 \cdot 10^1$	$1.9 \cdot 10^1$	$2.9 \cdot 10^1$	$4.8 \cdot 10^1$
System 2	F_{12} CPU	$7.6 \cdot 10^0$	$3.8 \cdot 10^1$	$9.8 \cdot 10^1$	$2.7 \cdot 10^2$	$4.5 \cdot 10^2$
	F_{12} GPU	$4.0 \cdot 10^0$	$9.1 \cdot 10^0$	$1.5 \cdot 10^1$	$2.4 \cdot 10^1$	$3.9 \cdot 10^1$
	F_1F_2 CPU	$3.9 \cdot 10^0$	$2.0 \cdot 10^1$	$4.8 \cdot 10^1$	$1.2 \cdot 10^2$	$2.6 \cdot 10^2$
	F_1F_2 GPU	$3.1 \cdot 10^0$	$7.3 \cdot 10^0$	$1.2 \cdot 10^1$	$1.8 \cdot 10^1$	$2.7 \cdot 10^1$

5.4 Overall performance

Next, we measure the computational times for the full algorithms. First, we consider Example 1 in Section 5.1.1 with the different sizes $n = 625$, 2500, 5625, 10000, 15625, and 22500, and the step size $h = 0.005$.

In Fig. 6 (left), we plot the ratio between the computing time of the CPU and of the GPU as a function of the problem size when applying the different methods to the arising DLE using both systems. The raw data can also be found in Table 2, except for the somewhat uninteresting case $n = 625$ which we omit due to space reasons. We observe that for small matrices the CPU implementation is less time-consuming than the GPU implementation. However, the GPU starts to pay off already for problem

Table 3 Computational costs of the algorithms applied to the DRE arising from Example 1 in Section 5.1.1, for different problem sizes (Wall-clock time, seconds)

	n	2500	5625	10000	15625	22500
System 1	$F_{12}F_3$ CPU	$8.5 \cdot 10^0$	$3.5 \cdot 10^1$	$8.8 \cdot 10^1$	$1.9 \cdot 10^2$	$4.1 \cdot 10^2$
	$F_{12}F_3$ GPU	$6.2 \cdot 10^0$	$1.5 \cdot 10^1$	$2.7 \cdot 10^1$	$5.0 \cdot 10^1$	$8.8 \cdot 10^1$
	$F_1F_2F_3$ CPU	$4.5 \cdot 10^0$	$2.1 \cdot 10^1$	$4.6 \cdot 10^1$	$1.1 \cdot 10^2$	$2.1 \cdot 10^2$
	$F_1F_2F_3$ GPU	$5.4 \cdot 10^0$	$1.2 \cdot 10^1$	$2.0 \cdot 10^1$	$3.1 \cdot 10^1$	$5.0 \cdot 10^1$
	$F_1F_3F_2$ CPU	$4.4 \cdot 10^0$	$2.1 \cdot 10^1$	$4.6 \cdot 10^1$	$1.1 \cdot 10^2$	$2.1 \cdot 10^2$
	$F_1F_3F_2$ GPU	$4.9 \cdot 10^0$	$1.1 \cdot 10^1$	$1.9 \cdot 10^1$	$2.9 \cdot 10^1$	$4.9 \cdot 10^1$
System 2	$F_{12}F_3$ CPU	$7.8 \cdot 10^0$	$3.8 \cdot 10^1$	$1.1 \cdot 10^2$	$2.7 \cdot 10^2$	$5.3 \cdot 10^2$
	$F_{12}F_3$ GPU	$4.8 \cdot 10^0$	$9.8 \cdot 10^0$	$1.6 \cdot 10^1$	$2.6 \cdot 10^1$	$4.0 \cdot 10^1$
	$F_1F_2F_3$ CPU	$4.1 \cdot 10^0$	$2.0 \cdot 10^1$	$4.8 \cdot 10^1$	$1.1 \cdot 10^2$	$2.5 \cdot 10^2$
	$F_1F_2F_3$ GPU	$3.9 \cdot 10^0$	$8.2 \cdot 10^0$	$1.3 \cdot 10^1$	$2.0 \cdot 10^1$	$2.9 \cdot 10^1$
	$F_1F_3F_2$ CPU	$3.9 \cdot 10^0$	$2.0 \cdot 10^1$	$4.8 \cdot 10^1$	$1.1 \cdot 10^2$	$2.5 \cdot 10^2$
	$F_1F_3F_2$ GPU	$3.5 \cdot 10^0$	$7.6 \cdot 10^0$	$1.2 \cdot 10^1$	$1.9 \cdot 10^1$	$2.7 \cdot 10^1$

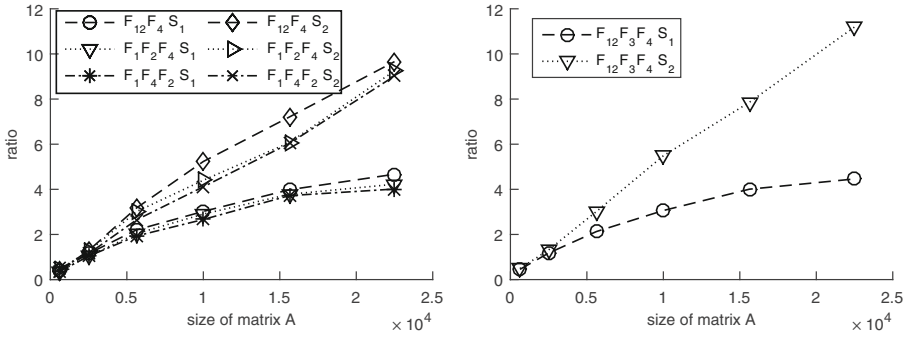

Fig. 7 Relative computational costs of the algorithms for the generalized DLE (left) and the generalized DRE (right) arising from Example 2 in Section 5.1.2, plotted versus the different problem sizes

sizes around $n = 2500$. For the largest test case $n = 22500$, we observe a speed-up of a factor 4.7 on System 1 and a factor of 11.7 on System 2 for the quadrature rule. For the splitting scheme, the speed-up is less but not by much. We note that these ratios are higher than the theoretical numbers which one might expect, but fully in line with the analysis in the previous section. We also remark here that the quadrature method is clearly more efficient than the splitting scheme, since the former method produces a much lower error than the latter while their computational costs are very similar.

A similar behavior can be seen in Fig. 6 (right), where we plot the GPU speed-up of the splitting schemes applied to the arising DRE. The break-even point is again around $n = 2500$, as seen in Table 3 which presents the raw data. For the largest test case, we again observe a factor 4.7 speed-up for the GPU implementation of the two-term splitting on System 1. On System 2, the corresponding number is 13.1.

Table 4 Computational costs of the algorithms applied to the generalized DLE arising from Example 2 in Section 5.1.2, for different problem sizes (Wall-clock time, seconds)

	n	2500	5625	10000	15625	22500
System 1	$F_{12}F_4$ CPU	$9.3 \cdot 10^0$	$3.6 \cdot 10^1$	$8.3 \cdot 10^1$	$2.1 \cdot 10^2$	$4.4 \cdot 10^2$
	$F_{12}F_4$ GPU	$8.1 \cdot 10^0$	$1.6 \cdot 10^1$	$2.8 \cdot 10^1$	$5.3 \cdot 10^1$	$9.4 \cdot 10^1$
	$F_1F_2F_4$ CPU	$7.7 \cdot 10^0$	$3.0 \cdot 10^1$	$6.5 \cdot 10^1$	$1.6 \cdot 10^2$	$3.2 \cdot 10^2$
	$F_1F_2F_4$ GPU	$7.4 \cdot 10^0$	$1.5 \cdot 10^1$	$2.2 \cdot 10^1$	$4.3 \cdot 10^1$	$7.5 \cdot 10^1$
	$F_1F_4F_2$ CPU	$8.5 \cdot 10^0$	$3.2 \cdot 10^1$	$7.1 \cdot 10^1$	$1.8 \cdot 10^2$	$3.7 \cdot 10^2$
	$F_1F_4F_2$ GPU	$8.2 \cdot 10^0$	$1.6 \cdot 10^1$	$2.7 \cdot 10^1$	$4.9 \cdot 10^1$	$9.2 \cdot 10^1$
System 2	$F_{12}F_4$ CPU	$9.2 \cdot 10^0$	$4.0 \cdot 10^1$	$1.0 \cdot 10^2$	$2.3 \cdot 10^2$	$5.0 \cdot 10^2$
	$F_{12}F_4$ GPU	$7.2 \cdot 10^0$	$1.3 \cdot 10^1$	$1.9 \cdot 10^1$	$3.2 \cdot 10^1$	$5.2 \cdot 10^1$
	$F_1F_2F_4$ CPU	$6.7 \cdot 10^0$	$3.0 \cdot 10^1$	$7.1 \cdot 10^1$	$1.8 \cdot 10^2$	$3.6 \cdot 10^2$
	$F_1F_2F_4$ GPU	$5.4 \cdot 10^0$	$1.0 \cdot 10^1$	$1.6 \cdot 10^1$	$2.9 \cdot 10^1$	$3.9 \cdot 10^1$
	$F_1F_4F_2$ CPU	$7.4 \cdot 10^0$	$3.3 \cdot 10^1$	$8.3 \cdot 10^1$	$1.9 \cdot 10^2$	$4.4 \cdot 10^2$
	$F_1F_4F_2$ GPU	$6.4 \cdot 10^0$	$1.2 \cdot 10^1$	$2.0 \cdot 10^1$	$3.2 \cdot 10^1$	$4.8 \cdot 10^1$

Table 5 Computational costs of the algorithms applied to the generalized DRE arising from Example 2 in Section 5.1.2, for different problem sizes (Wall-clock time, seconds)

	n	2500	5625	10000	15625	22500
System 1	$F_{12}F_3F_4$ CPU	$9.4 \cdot 10^0$	$3.7 \cdot 10^1$	$8.8 \cdot 10^1$	$2.2 \cdot 10^2$	$4.3 \cdot 10^2$
	$F_{12}F_3F_4$ GPU	$8.0 \cdot 10^0$	$1.7 \cdot 10^1$	$2.9 \cdot 10^1$	$5.5 \cdot 10^1$	$9.7 \cdot 10^1$
System 2	$F_{12}F_3F_4$ CPU	$8.7 \cdot 10^0$	$3.9 \cdot 10^1$	$1.1 \cdot 10^2$	$2.5 \cdot 10^2$	$5.5 \cdot 10^2$
	$F_{12}F_3F_4$ GPU	$6.6 \cdot 10^0$	$1.3 \cdot 10^1$	$1.9 \cdot 10^1$	$3.2 \cdot 10^1$	$4.9 \cdot 10^1$

Again, a speed-up of the implementation on the GPU is detected for these problems. The factors for the three-term splittings are both about 4.2 on System 1 and about 9 on System 2, which means that all the methods perform better than what might be expected at first glance, due to differently optimized underlying codebases.

We note that the fact that some of the schemes are faster than the others does not mean that they are more efficient, since their errors are also different. By plotting the errors in Fig. 2 against the computation times, one can observe that the three-term schemes are most efficient for all error levels when Q is relatively small compared to R^{-1}, while the two-term splitting is more efficient otherwise. This also holds in general for other problem sizes.

The GPU-based codes also exhibit better performance for the generalized matrix equations, as seen in Fig. 7 and Tables 4 and 5. We consider here the four schemes mentioned in Section 2 applied to the generalized DLE and DRE arising from Example 2 in Section 5.1.2. The break-even point is here slightly lower, but the maximal speed-up factors are similar to the previous examples.

Finally, we measure the computation times also for the two real-world examples. The left plot in Fig. 8 shows the results of applying the DLE methods to Example 3 in Section 5.1.3 for different problem sizes, and the right plot shows the DRE methods applied to Example 4 in Section 5.1.4. Tables 6 and 7 contain the respective raw data.

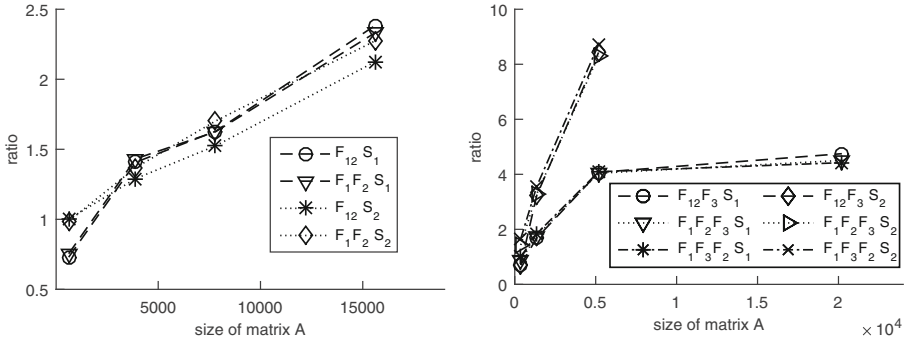

Fig. 8 Relative computational costs for Example 3 in Section 5.1.3 (left) and Example 4 in Section 5.1.4 (right). Due to time constraints, we could not test the largest problem size in Example 4 in Section 5.1.4 on System 2

Table 6 Computational costs of the algorithms applied to the DLE arising from Example 3 in Section 5.1.3, for different problem sizes (Wall-clock time, seconds)

	n	624	3900	7800	15600
System 1	F_{12} CPU	$2.6 \cdot 10^0$	$1.0 \cdot 10^1$	$1.9 \cdot 10^1$	$5.0 \cdot 10^1$
	F_{12} GPU	$3.5 \cdot 10^0$	$7.3 \cdot 10^0$	$1.1 \cdot 10^1$	$2.1 \cdot 10^1$
	F_1F_2 CPU	$2.5 \cdot 10^0$	$9.8 \cdot 10^0$	$1.8 \cdot 10^1$	$4.8 \cdot 10^1$
	F_1F_2 GPU	$3.3 \cdot 10^0$	$6.9 \cdot 10^0$	$1.1 \cdot 10^1$	$2.0 \cdot 10^1$
System 2	F_{12} CPU	$4.1 \cdot 10^0$	$1.1 \cdot 10^1$	$2.0 \cdot 10^1$	$4.6 \cdot 10^1$
	F_{12} GPU	$4.1 \cdot 10^0$	$8.9 \cdot 10^0$	$1.3 \cdot 10^1$	$2.2 \cdot 10^1$
	F_1F_2 CPU	$5.4 \cdot 10^0$	$1.1 \cdot 10^1$	$1.9 \cdot 10^1$	$4.7 \cdot 10^1$
	F_1F_2 GPU	$5.5 \cdot 10^0$	$8.0 \cdot 10^0$	$1.1 \cdot 10^1$	$2.1 \cdot 10^1$

The results are similar to the previous academic examples. In the DLE case, the CPU and GPU costs are comparable at $n = 624$ but at $n = 3900$ the GPU is more efficient, and at $n = 7800$, we already observe a speed-up of a factor 1.5. For the finest resolution, the speed-up is roughly a factor 2.4. The maximal speed-up is lower in this example, because the solution is of a higher rank than in the academic examples. This requires more work in the column compression step, which in turn performs SVD calculations. These are harder to parallelize than the other main sub-functions. We note, however, that as the problem dimension increases the solution rank increases only marginally. This means that for large enough problems the column compression cost will again be negligible, and the GPU speed-up will reach similar values as in

Table 7 Computational costs of the algorithms applied to the DRE arising from Example 4 in Section 5.1.4, for different problem sizes. Due to time constraints, we could not test the largest problem size on System 2 (wall-clock time, seconds)

	n	371	1357	5177	20209
System 1	$F_{12}F_3$ CPU	$3.2 \cdot 10^1$	$3.1 \cdot 10^2$	$2.7 \cdot 10^3$	$9.2 \cdot 10^4$
	$F_{12}F_3$ GPU	$4.6 \cdot 10^1$	$1.8 \cdot 10^2$	$6.7 \cdot 10^2$	$1.9 \cdot 10^4$
	$F_1F_2F_3$ CPU	$3.4 \cdot 10^1$	$3.1 \cdot 10^2$	$2.6 \cdot 10^3$	$7.9 \cdot 10^4$
	$F_1F_2F_3$ GPU	$3.8 \cdot 10^1$	$1.8 \cdot 10^2$	$6.4 \cdot 10^2$	$1.8 \cdot 10^4$
	$F_1F_3F_2$ CPU	$3.4 \cdot 10^1$	$3.2 \cdot 10^2$	$2.6 \cdot 10^3$	$7.5 \cdot 10^4$
	$F_1F_3F_2$ GPU	$3.4 \cdot 10^1$	$1.7 \cdot 10^2$	$6.2 \cdot 10^2$	$1.7 \cdot 10^4$
System 2	$F_{12}F_3$ CPU	$4.5 \cdot 10^1$	$3.4 \cdot 10^2$	$2.9 \cdot 10^3$	
	$F_{12}F_3$ GPU	$6.4 \cdot 10^1$	$1.1 \cdot 10^2$	$3.4 \cdot 10^2$	
	$F_1F_2F_3$ CPU	$3.7 \cdot 10^1$	$3.5 \cdot 10^2$	$2.8 \cdot 10^3$	
	$F_1F_2F_3$ GPU	$2.6 \cdot 10^1$	$1.1 \cdot 10^2$	$3.3 \cdot 10^2$	
	$F_1F_3F_2$ CPU	$3.5 \cdot 10^1$	$3.4 \cdot 10^2$	$2.7 \cdot 10^3$	
	$F_1F_3F_2$ GPU	$2.1 \cdot 10^1$	$9.6 \cdot 10^1$	$3.2 \cdot 10^2$	

the academic examples. We still want to emphasize that for the current largest test case the algorithm performs twice as good on the GPU as on the CPU, and this is with essentially no changes to the code.

In the DRE case, only the first problem size yields comparable costs for the CPU and GPU, and we observe speed-ups for all larger problem sizes. We obtain a speed-up of 4 already for $n = 5177$ and 4.7 for the largest test case on System 1. Due to time constraints, we only solve the first three problem sizes on System 2. For $n = 5177$, the speed-up is already more than 8.3 for all schemes, and we expect the ratio to level out similarly to what happens on System 1. As mentioned previously, even higher speed-up are expected when MATLAB supports solving equation systems with sparse system matrices and dense block right-hand sides.

6 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 on two different systems. In the considered problems of academical nature, the GPU code was faster than the pure CPU code by approximately a factor 3 already for matrices of size 10000 on System 1 and by a factor of 6 on System 2. This factor increases to over 4 and 10, respectively, for larger matrices of size 22500. The break-even point was around size 2500, which is well below what would be considered large-scale today. In the tested real-world applications, the gains were also in accordance with the more academic examples. 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 types. 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 off 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. (For example, take the most expensive subproblem as the "middle" term.) For the first question, we expected that it would not be beneficial to split off 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 Open access funding provided by Max Planck Society. The authors would like to thank the anonymous referees, whose critical and constructive comments greatly improved the manuscript. We are also grateful to Peter Kandolf for his assistance with the original `expleja` code.

Funding information This study is supported by the Austrian Science Fund (FWF)—project id:P27926 and by a scholarship of the Vizerektorat für Forschung, University of Innsbruck.

References

1. Abou-Kandil, H., Freiling, G., Ionescu, V., Jank, G.: Matrix Riccati equations in control and systems theory. Birkhäuser, Basel (2003)
2. Alonso-Mallo, I., Cano, B., Reguera, N.: Avoiding order reduction when integrating diffusion-reaction boundary value problems with exponential splitting methods. J. Comput. Appl. Math. **357**, 228–250 (2019)
3. Antoulas, A.C., Sorensen, D.C., Zhou, Y.: On the decay rate of Hankel singular values and related issues. Syst. Cont. Lett. **46**(5), 323–342 (2002)
4. Auer, N., Einkemmer, L., Kandolf, P., Ostermann, A.: Magnus integrators on multicore CPUs and GPUs. Comput. Phys. Comm. **228**, 115–122 (2018). https://doi.org/10.1016/j.cpc.2018.02.019
5. Auzinger, W., Koch, O., Thalhammer, M.: Defect-based local error estimators for high-order splitting methods involving three linear operators. Numer. Algorithm. **70**(1), 61–91 (2015). https://doi.org/10.1007/s11075-014-9935-8
6. Başar, T., Bernhard, P.: H^∞-optimal control and related minimax design problems. In: Systems & Control: Foundations & Applications. 2nd edn. Birkhäuser Boston, Inc., Boston (1995). https://doi.org/10.1007/978-0-8176-4757-5. A dynamic game approach
7. Baker, J., Embree, M., Sabino, J.: Fast singular value decay for Lyapunov solutions with nonnormal coefficients. SIAM. J. Matrix Anal. Appl. **36**(2), 656–668 (2015). https://doi.org/10.1137/140993867
8. Bell, N., Garland, M.: Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation (2008)
9. Benner, P., Breiten, T.: Low rank methods for a class of generalized Lyapunov equations and related issues. Numer. Math. **124**(3), 441–470 (2013). https://doi.org/10.1007/s00211-013-0521-0
10. Benner, P., Damm, T.: Lyapunov equations, energy functionals, and model order reduction of bilinear and stochastic systems. SIAM. J. Control Optim. **49**(2), 686–711 (2011)
11. Benner, P., Dufrechou, E., Ezzatti, P., Mena, H., Quintana-Ortí, E.S., Remón, A.: Solving sparse differential Riccati equations on hybrid CPU-GPU platforms. In: Gervasi, O., Murgante, B., Misra, S., Borruso, G., Torre, C.M., Rocha, A.M.A.C., Taniar, D., Apduhan, B.O., Stankova, E., Cuzzocrea, A. (eds.) Computational Science and Its Applications – ICCSA 2017: 17th International Conference, Trieste, Proceedings, Part I, pp. 116–132. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-62392-4_9
12. Benner, P., Ezzatti, P., Mena, H., Quintana-Ortí, E.S., Remón, A.: Solving differential Riccati equations on multi-GPU platforms. In: Proceedings of 11th International Conference on Computational and Mathematical Methods in Science and Engineering, pp. 178–188. CMMSE '11, Benidorm (2011)
13. Benner, P., Ezzatti, P., Mena, H., Quintana-Ortí, E.S., Remón, A.: Solving matrix equations on multi-core and many-core architectures. Algorithms **6**(4), 857–870 (2013). https://doi.org/10.3390/a6040857
14. Benner, P., Mena, H.: Rosenbrock methods for solving Riccati differential equations. IEEE Trans. Automat. Control **58**(11), 2950–2956 (2013). https://doi.org/10.1109/TAC.2013.2258495
15. Benner, P., Mena, H.: Numerical solution of the infinite-dimensional LQR problem and the associated Riccati differential equations. J. Numer. Math. **26**(1), 1–20 (2018). https://doi.org/10.1515/jnma-2016-1039

16. Benner, P., Ezzatti, P., Mena, H., Quintana-Ortí, E.S., Remón, A.: Unleashing CPU-GPU acceleration for control theory applications. In: Caragiannis, I., Alexander, M., Badia, R.M., Cannataro, M., Costan, A., Danelutto, M., Desprez, F., Krammer, B., Sahuquillo, J., Scott, S.L., Weidendorfer, J. (eds.) Euro-Par 2012: parallel processing workshops - BDMC, CGWS, HeteroPar, HiBB, OMHI, Paraphrase, PROPER, Resilience, UCHPC, VHPC, Rhodes Islands, Greece. Revised Selected Papers, Lecture Notes in Comput. Sci., vol. 7640, pp. 102–111. Springer (2012). https://doi.org/10.1007/978-3-642-36949-0
17. Benner, P., Saak, J.: A semi-discretized heat transfer model for optimal cooling of steel profiles. In: Benner, P., Mehrmann, V., Sorensen, D. (eds.) Dimension Reduction of Large-Scale Systems, Lect. Notes Comput. Sci. Eng., vol. 45, pp. 353–356. Springer, Berlin (2005). https://doi.org/10.1007/3-540-27909-1_19
18. Caliari, M., Kandolf, P., Ostermann, A., Rainer, S.: Comparison of software for computing the action of the matrix exponential. BIT Numer. Math. **54**(1), 113–128 (2014)
19. Caliari, M., Kandolf, P., Ostermann, A., Rainer, S.: The Leja method revisited: backward error analysis for the matrix exponential. SIAM. J. Sci. Comput. **38**(3), A1639–A1661 (2016)
20. Damm, T., Mena, H., Stillfjord, T.: Numerical solution of the finite horizon stochastic linear quadratic control problem. Numer. Lin. Alg. Appl. (2017)
21. De Leo, M., Rial, D., Sánchez de la Vega, C.: High-order time-splitting methods for irreversible equations. IMA, J. Numer. Anal. **36**(4), 1842–1866 (2016). https://doi.org/10.1093/imanum/drv058
22. Einkemmer, L., Ostermann, A.: Exponential integrators on graphic processing units. In: 2013 International Conference on High Performance Computing Simulation (HPCS), pp. 490–496. https://doi.org/10.1109/HPCSim.2013.6641458 (2013)
23. Einkemmer, L., Ostermann, A.: Overcoming order reduction in diffusion-reaction splitting. Part 1: Dirichlet boundary conditions. SIAM J. Sci. Comput. **37**(3), A1577–A1592 (2015). https://doi.org/10.1137/140994204
24. Einkemmer, L., Ostermann, A.: Overcoming order reduction in diffusion-reaction splitting. Part 2: Oblique boundary conditions. SIAM J. Sci. Comput. **38**(6), A3741–A3757 (2016). https://doi.org/10.1137/16M1056250
25. Farquhar, M.E., Moroney, T.J., Yang, Q., Turner, I.W.: GPU accelerated algorithms for computing matrix function vector products with applications to exponential integrators and fractional diffusion. SIAM J. Sci. Comput. **38**(3), C127–C149 (2016). https://doi.org/10.1137/15M1021672
26. Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., Koziris, N.: Understanding the performance of sparse matrix-vector multiplication. In: 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), pp. 283–292. https://doi.org/10.1109/PDP.2008.41 (2008)
27. Hansen, E., Ostermann, A.: High order splitting methods for analytic semigroups exist. BIT Numer. Math. **49**(3), 527–542 (2009). https://doi.org/10.1007/s10543-009-0236-x
28. Hansen, E., Stillfjord, T.: Convergence analysis for splitting of the abstract differential Riccati equation. SIAM J. Numer. Anal. **52**(6), 3128–3139 (2014). https://doi.org/10.1137/130935501
29. Hundsdorfer, W., Verwer, J.: Numerical solution of time-dependent advection-diffusion-reaction equations Springer Series in Computational Mathematics, vol. 33. Springer, Berlin (2003). https://doi.org/10.1007/978-3-662-09017-6
30. Ichikawa, A., Katayama, H.: Remarks on the time-varying H_∞ Riccati equations. Syst. Cont. Lett. **37**(5), 335–345 (1999)
31. Koskela, A., Mena, H.: Analysis of Krylov subspace approximation to large scale differential Riccati equations. arXiv:1705.07507 (2017)
32. Lang, N.: Numerical methods for large-scale linear time-varying control systems and related differential matrix equations. Dissertation, Technische Universität Chemnitz, Chemnitz (2017)
33. Lang, N., Mena, H., Saak, J.: On the benefits of the LDL^T factorization for large-scale differential matrix equation solvers. Linear Algebra Appl. **480**, 44–71 (2015). https://doi.org/10.1016/j.laa.2015.04.006
34. Mena, H., Ostermann, A., Pfurtscheller, L.M., Piazzola, C.: Numerical low-rank approximation of matrix differential equations. J. Comput. Appl. Math. **340**, 602–614 (2018)
35. Mena, H., Pfurtscheller, L.: An efficient SPDE approach for El Niño. Appl. Math. Comput. **352**, 146–156 (2019). https://doi.org/10.1016/j.amc.2019.01.071
36. Nakatsukasa, Y.: Gerschgorin's theorem for generalized eigenvalue problems in the Euclidean metric. Math. Comp. **80**(276), 2127–2142 (2011). https://doi.org/10.1090/S0025-5718-2011-02482-8

37. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008). https://doi.org/10.1145/1365490.1365500
38. Penland, C., Sardeshmukh, P.: The optimal growth of tropical sea surface temperature anomalies. J. Clim. **8**, 1999–2024 (1995)
39. Penzl, T.: Eigenvalue decay bounds for solutions of Lyapunov equations: the symmetric case. Syst. Cont. Lett. **40**, 139–144 (2000). https://doi.org/10.1016/S0167-6911(00)00010-4
40. Petersen, I.R., Ugrinovskii, V.A., Savkin, A.V.: Robust Control Design using H^∞ Methods. Springer, London (2000)
41. Reese, J., Zaranek, S.: GPU programming in MATLAB. Mathworks News & Notes, pp. 22–5. The MathWorks Inc, Natick (2012)
42. Saak, J.: Effiziente numerische Lösung eines Optimalsteuerungsproblems für die Abkühlung von Stahlprofilen. Diplomarbeit, Fachbereich 3/Mathematik und Informatik, Universität Bremen, D-28334 Bremen (2003)
43. Sorensen, D.C., Zhou, Y.: Bounds on eigenvalue decay rates and sensitivity of solutions to Lyapunov equations. Tech. Rep. TR02-07, Dept. of Comp. Appl. Math. Rice University, Houston (2002). Available online from https://scholarship.rice.edu/handle/1911/101987
44. Stillfjord, T.: Low-rank second-order splitting of large-scale differential Riccati equations. IEEE Trans. Automat. Control **60**(10), 2791–2796 (2015). https://doi.org/10.1109/TAC.2015.2398889
45. Stillfjord, T.: Adaptive high-order splitting schemes for large-scale differential Riccati equations. Numer. Algorithms. https://doi.org/10.1007/s11075-017-0416-8 (2017)