A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms

P. Benner[1], P. Ezzatti[2], D. Kressner, E.S. Quintana-Ortí[3] and A. Remón[3]

Research Report No. 2009-40
December 2009

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

[1]Fakultat fur Mathematik, TU Chemnitz, D-09107 Chemnitz, Germany
[2]Centro de Calculo-Instituto de la Computación, Universidad de la Republica, 11300 Montevideo, Uruguay
[3]Dpto. de Ingeniera y Ciencia de Computadores, Universidad Jaime I, 12071 Castellón, Spain

A Mixed-Precision Algorithm for the Solution of Lyapunov Equations on Hybrid CPU-GPU Platforms

Peter Benner[a], Pablo Ezzatti[b], Daniel Kressner[c], Enrique S. Quintana-Ortí[d], Alfredo Remón[d]

[a]*Fakultät für Mathematik, TU Chemnitz, D-09107 Chemnitz (Germany)*
[b]*Centro de Cálculo-Instituto de la Computación, Universidad de la República, 11.300–Montevideo (Uruguay)*
[c]*Seminar für Angewandte Mathematik, ETHZ, CH-8092 Zürich (Switzerland)*
[d]*Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaime I, 12.071-Castellón (Spain)*

Abstract

We describe a hybrid Lyapunov solver based on the matrix sign function that accelerates the intensive parts of the computation using a graphics processor (GPU) while executing the remaining operations in a general-purpose multi-core processor. The initial stage of the iterative solver operates in single-precision arithmetic, to exploit the many-core parallelism of current GPUs, returning a full-rank factor to the solution of the equation. To improve this approximate solution, the second stage consists of an efficient iterative refinement procedure that allows to cheaply recover full double-precision accuracy. The combination of these two stages results in a mixed-precision algorithm, that exploits the capabilities of both general-purpose multi-core processors and many-core GPUs, overlapping critical computations.

Experiments using a platform equipped with two Intel Xeon QuadCore processors and an Nvidia Tesla C1060 show the efficiency of this approach to solve Lyapunov equations arising in practical model reduction applications: compared with a classical implementation that exploits the parallelism of a general-purpose processor using a multi-threaded implementation of BLAS and operates in double-precision, our hybrid algorithm delivers 4.24–6.46 speed-ups while attaining the same accuracy in the solution.

Key words: Lyapunov equations, matrix sign function, graphics processors, multi-core processors, control theory

1. Introduction

We consider the solution of a *Lyapunov matrix equation* with factored right-hand side:

$$AX + XA^T \;=\; -BB^T, \qquad (1)$$

where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$ are the coefficient matrices, and $X \in \mathbb{R}^{n\times n}$ is the desired solution. Such matrix equations play a key role in several applications from control theory: SVD-based methods for balanced truncation model reduction [1, 2], Newton methods for the algebraic Riccati equation associated with linear-quadratic optimal control problems [3], stabilization methods and stability tests for linear dynamical systems as well as the computation of the H_2-norm of stable linear control systems [4, 5] all require the solution of one or more Lyapunov equations. Complex control systems or discretizations of partial differential equations easily lead to coefficients of order $n = 10^4 - 10^5$ or even larger. Note, however, that m, the number of columns of B, is typically much smaller than n in these applications [6]. In what follows, we will assume that A is a c-stable matrix, that is, all eigenvalues of A have negative real part. This implies that X is a symmetric positive semi-definite matrix; it is also necessary for the viability of the sign function-based methods discussed in this paper. This stability condition is usually satisfied by the equations arising in model reduction and optimal control problems.

There are several approaches to solve Lyapunov equations. Transforming (1) into an equivalent $n^2 \times n^2$ linear system via the Kronecker product is only feasible for very small-scale equations [7]. An alternative is to first reduce A to real Schur form via the QR algorithm, solve the associated reduced Lyapunov equation, and finally apply a back-transformation to obtain the solution to the original equation [8, 9]. Although this can be implemented in message-passing platforms to solve Lyapunov equations, it has two main drawbacks: first, the main computational task in the procedure, the computation of the real Schur form, is difficult to implement efficiently in current

Email addresses: benner@mathematik.tu-chemnitz.de (Peter Benner), pezzatti@fing.edu.uy (Pablo Ezzatti), daniel.kressner@sam.math.ethz.ch (Daniel Kressner), quintana@icc.uji.es (Enrique S. Quintana-Ortí), remon@icc.uji.es (Alfredo Remón)

message-passing architectures and these difficulties are likely to carry over to a data-parallel implementation (see [10] for some recent work and [11, 12] for highly efficient triangular Lyapunov solvers); and second, there is currently no variant of this method that can exploit the usual low-rank structure of the Cholesky factor of the solution X. Thus, in the past, large Lyapunov equations have been tackled using message-passing parallel solvers based on the matrix sign function, which were then executed on clusters with a moderate number of nodes/processors [13]. The result of this effort was the message-passing library `PLiC` [14] and subsequent libraries for model reduction (`PLiCMR`, see [15]) and optimal control (`PLiCOC`, see [16]). Using these libraries, a few years ago 16–32 processors showed to provide enough computational power to solve Lyapunov equations with n up to $10,000$ and with satisfactorily parallel efficiency.

Recent work on the implementation of BLAS and the major factorization routines for the solution of linear systems [17, 18, 19, 20, 21] has demonstrated the potential of graphics processors (GPUs) to yield high performance on dense linear algebra operations that can be cast in terms of matrix-matrix products. Following the evolution of GPUs and the increase in the number of cores of current general-purpose processors, in this paper we evaluate the impact of these new architectures on the matrix sign function-based solvers. Specifically, this paper provides the following contributions:

- We implement and evaluate a hybrid CPU-GPU matrix sign function-based solver that off-loads the computationally intensive parts to the GPU. Our approach employs a "sequential" version of the message-passing codes in the `PLiC` library, and extracts parallelism from calls to tuned multi-threaded implementations of the BLAS (*Basic Linear Algebra Subprograms*) for the general-purpose multi-core processor (Intel MKL) and the GPU (CUBLAS).

- To exploit the different speed between single-precision and double-precision real arithmetic of current architectures (a factor 8:1 in GPUs and 2:1 in general-purpose cores), we propose a new iterative refinement procedure to recover full double-precision accuracy from an approximate solution to the Lyapunov equation. The iterative procedure could also be of use in other situations, where a cheap but rather inaccurate solver for Lyapunov equations is available.

- The experimental results on a current general-purpose multi-core processor and a GPU provide strong evidence that this is a valid platform to deal with Lyapunov equations which, only a few years ago, would have required the use of a distributed-memory cluster. This result is particularly important in control theory applications like, e.g., model reduction and optimal control, as control engineers rarely make use of distributed computing platforms to address their computational problems.

The rest of the paper is structured as follows. In Section 2 we describe the matrix-sign function approach for the solution of Lyapunov equations; this section also contains the two first major contributions in our paper: the hybrid CPU-GPU implementation of the solver and the iterative refinement procedure. This is followed by experimental results in Section 3, while concluding remarks and open questions close the paper in Section 4.

2. Lyapunov Solvers

2.1. Solution of Lyapunov equations via the matrix sign function

Consider a matrix $M \in \mathbb{R}^{n\times n}$ with no eigenvalues on the imaginary axis, and let

$$M = T^{-1} \begin{pmatrix} J_- & 0 \\ 0 & J_+ \end{pmatrix} T, \tag{2}$$

be its Jordan decomposition, where $J_- \in \mathbb{R}^{j\times j}$ and $-J_+ \in \mathbb{R}^{(n-j)\times(n-j)}$ are both c-stable [22]. The *matrix sign function* of M is then defined as

$$\operatorname{sign}(M) = T^{-1} \begin{pmatrix} -I_j & 0 \\ 0 & I_{n-j} \end{pmatrix} T, \tag{3}$$

where I denotes the identity matrix of the order indicated by the subscript. There are simple iterative schemes for the computation of the sign function. Among these, the Newton iteration, given by

$$\begin{aligned} M_0 &\leftarrow M, && (4)\\ M_{k+1} &\leftarrow \frac{1}{2}(M_k + M_k^{-1}), \quad k = 0, 1, 2, \ldots, && (5) \end{aligned}$$

is specially appealing for its simplicity, efficiency, parallel performance, and asymptotic quadratic convergence [23, 24].

Now consider the standard Lyapunov equation

$$AX + XA^T \;=\; -Q, \tag{6}$$

where $A, Q \in \mathbb{R}^{n\times n}$, $Q = Q^T$, and $X = X^T \in \mathbb{R}^{n\times n}$ is the desired solution. Assuming A is c-stable, we can apply the Newton iteration in (4)–(5) to

$$M \;=\; \begin{bmatrix} A^T & 0 \\ Q & -A \end{bmatrix}, \tag{7}$$

and it can be shown that, in this case,

$$M_\infty \;=\; \lim_{k\to\infty} M_k \;=\; \begin{bmatrix} -I_n & 0 \\ X & I_n \end{bmatrix}; \tag{8}$$

see [25]. In practice, rather than performing one iteration with $2n \times 2n$ matrices M_k, it is cheaper to consider two iterations with $n \times n$ matrices, leading to the following algorithm:

```
Algorithm GECLNW: Basic Newton iteration
$A_0 \leftarrow A$, $Q_0 \leftarrow Q$.
for $k = 0, 1, 2, \ldots$ until convergence
    $A_{k+1} \leftarrow \frac{1}{2c_k}\left(A_k + c_k^2 A_k^{-1}\right)$
    $Q_{k+1} \leftarrow \frac{1}{2c_k}\left(Q_k + c_k^2 A_k^{-1} Q_k A_k^{-T}\right)$
```

In this algorithm the scalar c_k is chosen to accelerate the convergence of the iteration: $\lim_{k\to\infty} A_k = \text{sign}\,(A) = -I_n$, $\lim_{k\to\infty} Q_k = X$. Although *determinantal scaling* [23] is frequently used to determine this value, following a recent study in [26] we prefer the *Euclidian (2-norm) scaling*:

$$c_k \leftarrow \sqrt{\frac{\|A\|_2}{\|A^{-1}\|_2}}. \tag{9}$$

To avoid the expensive computation of the matrix 2-norm, we use instead the (sometimes quite rough) approximations $\|M\|_2 \approx \sqrt{\|M\|_1 \|M\|_\infty}$ or $\|M\|_2 \approx \|M\|_F$, which are much cheaper to compute. The convergence of the iteration is checked as

$$\|A_k + I_n\|_F < \tau_{\text{iter}}. \tag{10}$$

Given the quadratic convergence of this iteration and to avoid stagnation of the procedure, we set $\tau_{\text{iter}} = 10 \cdot \sqrt{n \cdot \varepsilon}$, where ε denotes the machine precision, and perform two additional iterations once this convergence criterion is satisfied.

Lyapunov equations arising in control theory problems frequently exhibit a factored right-hand side matrix, as in (1), implying a positive semidefinite solution X. Hence, there exists a full-rank factorization $X = LL^T$ where $\operatorname{rank}(L) = \operatorname{rank}(X)$. Practical applications often require the factor L rather than the explicit full solution X, so that the following factored iteration becomes rather appealing:

Algorithm `GECLNC`**:** Newton iteration for the factored solution

```
$A_0 \leftarrow A$, $B_0 \leftarrow B$
for $k = 0, 1, 2, \ldots$ *until convergence*
    $A_{k+1} \leftarrow \frac{1}{2c_k}\left(A_k + c_k^2 A_k^{-1}\right)$
    $B_{k+1} \leftarrow \frac{1}{\sqrt{2c_k}}[\, B_k,\ c_k A_k^{-1} B_k \,]$
```

Here, $\lim_{k\to\infty} B_k B_k^T = \lim_{k\to\infty} Q_k = X$, so that B_∞ is a full-rank factor of the solution matrix X. The computational cost per iteration of this algorithm is $2n^3 + 2n^2\bar{m}_k$ flops (floating-point arithmetic operations), where $\bar{m}_k$ is the number of columns of B_k. Note that $\bar{m}_k$ and hence the work space required to store B_k is doubled in each iteration step. Fortunately, the dimensions of this workspace (as well as the computational cost of the iterative scheme) can be significantly reduced by computing in each iteration step a rank-revealing QR (RRQR) factorization [22] of B_{k+1}^T such that

$$B_{k+1}^T \Pi_{k+1} = U_{k+1} \begin{bmatrix} R_{k+1} \\ 0 \end{bmatrix}, \qquad (11)$$

where Π_{k+1} is a permutation matrix, U_{k+1} is orthogonal, and R_{k+1} is upper triangular. By truncating negligible entries, the order of R_{k+1} can often be made much smaller than $\bar{m}_k$. Since

$$\begin{aligned} B_{k+1}B_{k+1}^T &= \Pi_{k+1}R_{k+1}^T U_{k+1}^T U_{k+1} R_{k+1} \Pi_{k+1}^T = (\Pi_{k+1}R_{k+1}^T)(R_{k+1}\Pi_{k+1}) \\ &= \bar{R}_{k+1}^T \bar{R}_{k+1}, \end{aligned} \qquad (12)$$

it is possible to replace B_{k+1} with $\bar{R}_{k+1}^T$. The cost of this compression procedure using, e.g., the QR factorization with column pivoting to obtain an RRQR factorization of B_{k+1}^T) is $(8n\bar{m}_k - 2\bar{m}_{k+1}(n + 2\bar{m}_k) + 4\bar{m}_{k+1}^2/3)\bar{m}_{k+1}$ flops, noting that U_{k+1} need not be accumulated.

2.2. Hybrid CPU-GPU implementation of the Newton iteration for the factored solution

Algorithm `GECLNC` is basically composed of the following computational kernels:

– Matrix inversion A_k^{-1}.

– Matrix product $A_k^{-1}B$.

– QR factorization with column pivoting (RRQR factorization).

– Other minor operations related with the computation of c_k, like $\|A_k\|_{1/\infty}$, $\|A_k^{-1}\|_{1/\infty}$, and the convergence test.

We next discuss the efficient implementation of the major kernels on a CPU-GPU platform and some other related implementation details.

2.2.1. Matrix inversion via Gauss-Jordan elimination

The traditional procedure to invert a matrix A (as implemented, e.g., in LAPACK [27]) first computes an LU factorization with partial pivoting: $PA = LU$, where P is a permutation matrix, L is lower unit triangular and U is upper triangular. This is followed by inverting the triangular factor U^{-1}, solving the triangular linear system $YL = U^{-1}$ for Y and, finally, obtaining the inverse from the permutation $A^{-1} = YP$.

The first three stages of the procedure described above can be cast in terms of BLAS-3 kernels, which deliver high performance on current general-purpose multi-core processors, while the final permutation stage presents a negligible cost compared with the other three. However, our implementation of matrix inversion on a CPU-GPU platform and the experimental evaluation in [28] revealed that a procedure based on Gauss-Jordan elimination (GJE) delivers higher performance on this type of platforms due to the regularity/larger dimensions of the GPU kernels involved in this method. We next describe an improved variant of the inversion procedure in [28], which incorporates look-ahead [29] to allow computations to proceed concurrently in the CPU and in the GPU.

Consider, for simplicity, that no pivoting is required during the inversion of A. The following (unblocked) algorithm overwrites A with its inverse:

Algorithm GEINGJ: Matrix inversion via GJE

```
for $k = 1, 2, \ldots, n$
    Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^T & \alpha_{11} & a_{12}^T \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$ where $A_{00} \in \mathbb{R}^{(k-1)\times(k-1)}$, $\alpha_{11} \in \mathbb{R}$
    $p \leftarrow \alpha_{11}$
    $[a_{01},\ \alpha_{11},\ a_{12}]^T \leftarrow -[a_{01},\ 0,\ a_{12}]^T \ / \ p$
    $A \leftarrow A + [a_{01},\ \alpha_{11},\ a_{12}]^T \cdot \left[a_{10}^T,\ \alpha_{11},\ a_{21}^T\right]$
    $\left[a_{10}^T,\ \alpha_{11},\ a_{21}^T\right] \leftarrow \left[a_{10}^T,\ 1,\ a_{21}^T\right] \ / \ p$
```

A blocked variant of this algorithm, with algorithmic block size b, is obtained by aggregating parts of the computations in terms of BLAS-3 operations (for simplicity, we assume n to be an integer multiple of b):

Algorithm GEINGJ_BLK: Matrix inversion via GJE (blocked)

```
for $k = 1, 2, \ldots, n/b$
    Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$ where $A_{00} \in \mathbb{R}^{(k-1)b\times(k-1)b}$, $A_{11} \in \mathbb{R}^{b\times b}$
    $[A_{01},\ A_{11},\ A_{21}]^T \leftarrow$ GEINGJ($[A_{01},\ A_{11},\ A_{21}]^T$)
    $A_{00} \leftarrow A_{00} + A_{01}A_{10}$
    $A_{20} \leftarrow A_{20} + A_{21}A_{10}$
    $A_{10} \leftarrow A_{11}A_{10}$
    $A_{02} \leftarrow A_{02} + A_{01}A_{12}$
    $A_{22} \leftarrow A_{22} + A_{21}A_{12}$
    $A_{12} \leftarrow A_{11}A_{12}$
```

The cost of these inversion procedures is $2n^3$ flops; i.e., the same as the cost of inverting a matrix via Gaussian elimination (LU factorization).

Suppose that A initially resides in the GPU (memory space). The hybrid CPU-GPU implementation of the blocked GJE inversion procedure in [28] performs the computations corresponding to `GEINGJ` in the CPU while all remaining operations (matrix-matrix products) are computed on the GPU. At the beginning of each iteration, $\left[A_{01}^T,\ A_{11}^T,\ A_{21}^T\right]^T$ is transferred from the GPU to the CPU (memory space), factored there, and the results are sent back to the GPU before the remaining computations proceed. We improve this procedure with a look-ahead strategy (of dimension b) that enables overlapping of the computations performed by CPU and GPU. In particular, consider the partitioning

$$\left[\begin{array}{c} A_{02} \\ \hline A_{12} \\ \hline A_{22} \end{array}\right] \rightarrow \left[A_L,\ A_R\right],$$

with $A_L \in \mathbb{R}^{n\times b}$. The look-ahead variant updates first the columns of A_L on the GPU and immediately transfers the contents of this block to the CPU. Acting in this manner, the CPU can factor A_L (in advance for the next iteration) concurrently with the rest of the computations in the GPU (for the current iteration).

2.2.2. Matrix product

Due to the combined effects of $m \ll n$ in practical control applications, the rapid convergence of the Newton iteration, the early stop of this iteration and transition to the iterative refinement procedure, and the benefits of column compression, we expect the number of columns of the sequence B_k to stay quite low. On the other hand, as will be described in Section 2.3, the sequence of inverses A_k^{-1} needs to be saved to be used later, during the iterative refinement. As the amount of memory on the GPU is limited, we transfer A_k^{-1} at each iteration to the CPU and store it there. Therefore, the matrix products $A_k^{-1}B_k$ can likely be computed as efficiently in the CPU as in the GPU, and the execution time of this step is expected to be small compared with that of computing the matrix inverse. In our implementation we compute these matrix products on the CPU. Should the execution time of the matrix product become significant, we could in principle overlap the inversion of A_{k+1} using one core of the CPU (for the factorization of A_L; see above) and the GPU with the computation of $A_k^{-1}B_k$ using the remaining cores of the CPU. Our experimental results will demonstrate that this is not necessary.

2.2.3. Compression via the QR factorization with column pivoting

Currently, there is no data-parallel implementation of an RRQR factorization for GPUs available. Thus, the only option is to compute the compressed factor on the CPU, using e.g. the LAPACK routine `GEQP3` for the QR factorization with column pivoting, and extract parallelism by calling multi-threaded implementations of BLAS. We employ a cheap heuristic to estimate the rank of the triangular factor resulting from this QR factorization, using a rank tolerance $\tau_{\text{rank}} = 10 \cdot \sqrt{n} \cdot \varepsilon$.

On the other hand, the compression procedure is fairly expensive and, therefore, from a computational viewpoint it is only recommended when m is large or the number of steps of the Newton iterative scheme becomes large. Whether the computation of the corresponding factorization reduces the overall cost of the iteration depends on the problem at hand. We will discuss this issue further during the experimentation.

2.3. Iterative refinement

The solution computed by the hybrid CPU-GPU implementation described in Section 2.2 may be less accurate than desired. First, computations on the GPU are typically performed in single-precision arithmetic but double-precision accuracy is the standard in scientific computing. Second, as we will see in the numerical experiments, it pays off to stop the Newton iteration `GECLNC` prematurely and attain high accuracy by other means.

2.3.1. General idea

In the following, we describe a procedure for refining factored approximations to solutions of Lyapunov equations. For this purpose, we assume the availability of an inexact solver `ApproxLyap` that produces approximations of low rank to the solution of $AX + XA^T = -BB^T$ for any right-hand side factor B: $X \approx LL^T$ with $L = \texttt{ApproxLyap}(B)$. In the scope of this paper, `ApproxLyap` stands for routine `GECLNC` performed in single-precision arithmetic and/or stopped prematurely. However, it is worth noting that our procedure is applicable to other settings, e.g., for refining the result of a sign function iteration performed in hierarchical matrix arithmetic [30, 31].

Let $L_0 = \texttt{ApproxLyap}(B)$. To improve the approximation quality of the corresponding approximate solution $X_0 = L_0L_0^T$, we construct a correction based on the residual

$$\mathcal{R}(L_0) = AL_0L_0^T + L_0L_0^TA^T + BB^T.$$

Note that the residual is symmetric but generally indefinite. In a standard iterative refinement scheme [32], one would simply solve a Lyapunov equation with right hand side $-\mathcal{R}(L_0)$ to obtain the correction term. However, this is not possible in our framework as the solver behind `ApproxLyap` assumes the right-hand side to be negative semi-definite. To overcome this limitation, we decompose

$$\mathcal{R}(L_0) = B_+B_+^T - B_-B_-^T. \tag{13}$$

The computational aspects of this decomposition will be discussed in Section 2.3.2 below.

The decomposition (13) results in two Lyapunov equations

$$AX_+ + X_+A^T = -B_+B_+^T, \quad A(-X_-) + (-X_-)A^T = -B_-B_-^T,$$

with $X_c = X_+ + X_-$ solving the correction equation $AX_c + X_cA^T = -\mathcal{R}(L_0)$. Approximating $L_+ = \texttt{ApproxLyap}(B_+)$ and $L_- = \texttt{ApproxLyap}(B_-)$, the corrected solution therefore takes the form $X_1 = L_0L_0^T + L_+L_+^T - L_-L_-^T$. Similar to (13), we decompose X_1 into positive/negative semi-definite parts:

$$X_1 = L_1L_1^T - \widetilde{L}_-\widetilde{L}_-^T, \quad \widetilde{L}_-^T L_1 = 0. \tag{14}$$

The orthogonality constraint $\widetilde{L}_-^T L_1 = 0$ ensures that $L_1L_1^T$ is the best symmetric positive semi-definite approximation to X_1. We can therefore neglect the term $\widetilde{L}_-$ and continue the iteration with L_1 as new iterate. This leads to the following algorithm.

Algorithm `LYAPREF`**:** Iterative refinement for the factored solution

```
for $k = 0, 1, 2, \dots$ *until convergence*
    $\mathcal{R}(L_k) \leftarrow AL_kL_k^T + L_kL_k^TA^T + BB^T$
    Decompose $\mathcal{R}(L_k) \rightarrow B_+B_+^T - B_-B_-^T$
    $L_+ \leftarrow$ ApproxLyap($B_+$),    $L_- \leftarrow$ ApproxLyap($B_-$)
    $X_{k+1} \leftarrow L_kL_k^T + L_+L_+^T - L_-L_-^T$
    Decompose $X_{k+1} \rightarrow L_{k+1}L_{k+1}^T - \widetilde{L}_-\widetilde{L}_-^T$
```

Note that, as explained in the next section, the matrices $\mathcal{R}(L_k)$ and X_{k+1} are never explicitly formed. The iteration is stopped as soon as the relative residual $\|\mathcal{R}(L_k)\|_F / \|L_kL_k^T\|_F$ is below a user-defined tolerance. As explained in Chapter 12 of [32], `LYAPREF` attains double-precision accuracy even with low accuracy in `ApproxLyap` provided that $\mathcal{R}(L_k)$ is formed in double-precision and the Lyapunov equation is not extremely ill-conditioned. There is no need to determine the decomposition of $\mathcal{R}(L_k)$ to high relative accuracy.

2.3.2. Decomposition into positive/negative semi-definite parts

In principle, the decomposition (13) can be directly obtained from a spectral decomposition of $\mathcal{R}(L_k) = AL_kL_k^T + L_kL_k^TA^T + BB^T$. However, from a computational point of view, neither the explicit computation nor the spectral decomposition of $\mathcal{R}(L_k)$ is desirable. To develop a more efficient approach, we rewrite

$$\mathcal{R}(L_k) = [L_k, AL_k, B] \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} [L_k, AL_k, B]^T =: F \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} F^T$$

and compute the (economic) QR factorization $F = UT$, where the columns of U are orthonormal and T is upper triangular. Then we compute a spectral decomposition of the (significantly smaller) matrix

$$T \begin{bmatrix} 0 & I & 0 \\ I & 0 & 0 \\ 0 & 0 & I \end{bmatrix} T^T = [Q_+, Q_-, Q_0] \begin{bmatrix} \Lambda_+ & 0 & 0 \\ 0 & \Lambda_- & 0 \\ 0 & 0 & \Lambda_0 \end{bmatrix} [Q_+, Q_-, Q_0]^T,$$

where Λ_+ contains the positive eigenvalues, Λ_- the negative eigenvalues, and Λ_0 the zero/negligible eigenvalues of $\mathcal{R}(L_k)$. As explained above, we can be generous in neglecting eigenvalues that are relatively small. In our experiments, eigenvalues of magnitude below $\tau_{\text{eig}} = 10^{-4} \cdot \max(\|\Lambda_+\|_1, \|\Lambda_-\|_1, \|\Lambda_0\|_1)$ were neglected.

The desired decomposition (13) is obtained by setting

$$L_+ = UQ_+\sqrt{\Lambda_+}, \quad L_- = UQ_-\sqrt{-\Lambda_-}.$$

Note that $L_+^TL_- = 0$ and hence the decomposition (14) can be obtained in an analogous way.

2.3.3. Combination with `GECLNC`

Every iteration of Algorithm `LYAPREF` requires two applications of the approximate Lyapunov solver `ApproxLyap`, which in our setting corresponds to running Algorithm `GECLNC` twice. However, the computational cost of `GECLNC`

Processors	#cores	Freq. (GHz)	L2 (MB)	Memory (GB)	Single/double-precision peak performance (GFLOPS)
Intel Xeon	8	2.3	12	8	149.1/74.6
Nvidia Tesla	240	1.3	–	4	933.0/78.0

Table 1: Hardware employed in the experiments.

can be significantly reduced if the iterates $A_1^{-1}, A_2^{-1}, \ldots$ are already available. We therefore propose to precompute and store $A_1^{-1}, \ldots, A_{\bar{k}}^{-1}$ for a fixed number $\bar{k}$ of Newton iterations. In the numerical experiments, we will see that $\bar{k}$ can be chosen quite small and hence the additional storage requirements remain limited.

3. Experimental Results

In this section we evaluate the numerical accuracy and parallel performance of the Lyapunov solvers as well as the performance of the basic matrix inversion kernels. The target platform consists of two Intel Xeon QuadCore processors connected to an Nvidia Tesla C1060 via a PCI-e bus; see Table 1 for details. We measured the performance using two different implementations of the BLAS, GotoBLAS [33] (version 1.26) and Intel MKL [34] (version 10.1), for the general-purpose processor, and Nvidia CUBLAS [35] (version 2.1) for the GPU. As the differences encountered between GotoBLAS and MKL were minor, we only report results for the second. We set `OMP_NUM_THREADS=8` so that one thread is employed per core in the parallel execution of the MKL routines in the two Intel Xeon QuadCore processors.

3.1. Matrix inversion

We first evaluate three parallel multi-threaded variants to compute the inverse of a matrix:

- `LAPACK+CPU`: LAPACK-based inversion procedure, with all computations carried out by the CPU and parallelism extracted by using a multi-threaded implementation of BLAS.

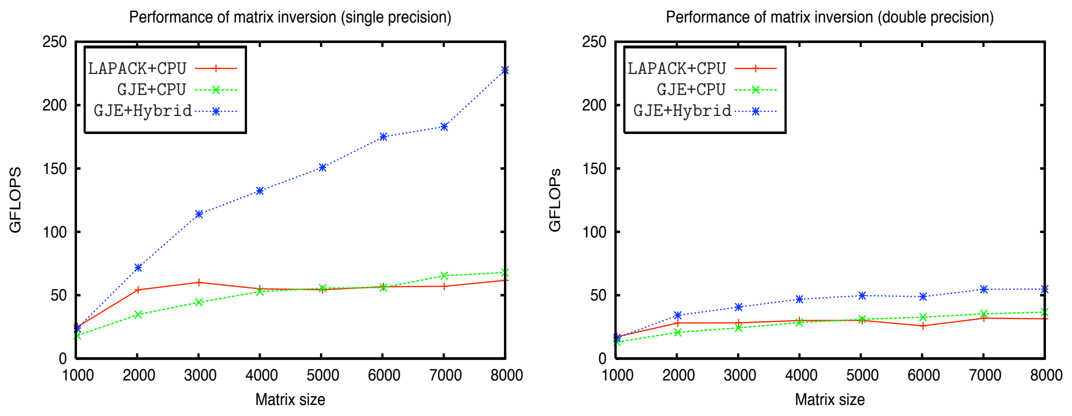

Figure 1: Performance of the matrix inversion variants using single and double-precision arithmetic (left and right, respectively).

- `GJE+CPU`: Matrix inversion via GJE (see Algorithm `GEING_BLK`), also with all computations performed by the CPU and parallelism extracted from a multi-threaded of BLAS.
- `GJE+Hybrid`: Matrix inversion via GJE with look-ahead, with most of the computations performed in the GPU, as described in Section 2.2.1.

Figure 1 reports the GFLOPS (10^9 flops per second) rate attained by the different implementations of the inversion codes operating on matrices with sizes 1,000, 2,000,. . . , 8,000. For the two matrix inversion variants based on GJE (`GJE+CPU` and `GJE+Hybrid`), several algorithmic block sizes were tested (parameter b in Algorithm `GEINGJ_BLK`); for simplicity, the results correspond to those obtained with the optimal block size. The timings reported for `GJE+Hybrid` include the cost of initially transferring the matrix to the GPU and retrieving the results to the CPU memory space once the matrix inverse is computed. For this particular variant, we augmented the matrix with a few more rows/columns so that its dimension was an integer multiple of 32. For large matrices, this only increases the number of operations performed during the matrix inversion slightly, but allows a much faster execution on the GPU; see, e.g., [17]. Nevertheless, the GFLOPS rate was always based on the dimension of the non-augmented matrix.

The results in the figure show a highest performance of 61, 67 and 227 GFLOPS for variants `LAPACK+CPU`, `GJE+CPU` and `GJE+Hybrid`, respectively, in single-precision arithmetic; i.e., the GPU variant is roughly 3.5 times faster than the CPU codes for the largest problem sizes. These performances decrease by a factor of 2 in the CPU and a factor of 4 in the GPU when double-precision arithmetic is employed. It is also important to notice that, while the performance of the CPU codes is a flat line already for problems of dimension 2,000, the GPU can potentially deliver higher performance in single-precision for problem dimensions larger than those reported in the figure.

3.2. Solution of Lyapunov equations

In the following experiments, we evaluate the performance of the Lyapunov solvers and iterative refinement applying the previous variants of the matrix inversion kernel during the solution of two Lyapunov equations. These problems are associated with two of the examples of the Oberwolfach model reduction benchmark collection at the University of Freiburg[1]:

- `STEEL_I`: This model arises in a manufacturing method for steel profiles [36]. The goal is to design a control that yields moderate temperature gradients when the rail is cooled down. The mathematical model corresponds to the boundary control for a 2-D heat equation. A finite element discretization, followed by adaptive refinement of the mesh results in the examples in this benchmark.

- `FLOW_METER`: This a 2-D model of an anemometer-like structure mainly consisting of a tube and a small heat source [37]. The Dirichlet boundary conditions are applied to the original system. The reference temperature is set to 300 K, and Dirichlet boundary conditions as well as initial conditions are set to 0 with respect to the reference.

The dynamical system in these examples is given by a tuple $(\tilde{E}, \tilde{A}, \tilde{B}, \tilde{C})$, where the pair $\tilde{E}, \tilde{A} \in \mathbb{R}^{n\times n}$ defines the state matrix pencil, $\tilde{B} \in \mathbb{R}^{n\times m}$ the input matrix, and $\tilde{C} \in \mathbb{R}^{p\times n}$ the output matrix. We transform the system into a standard one by considering instead $(I_n, \tilde{E}^{-1}\tilde{A}, \tilde{E}^{-1}\tilde{B}, \tilde{C})$, so that we

[1]`http://www.imtek.de/simulation/benchmark/`.

Example	n	m	p
STEEL_I	5,177	7	6
FLOW_METER	9,669	1	5

Table 2: Dimensions of the dynamical systems employed in the evaluation of the Lyapunov solvers.

solve the Lyapunov equation

$$\begin{array}{rcll} (\tilde{E}^{-1}\tilde{A})X + X(\tilde{E}^{-1}\tilde{A})^T & = & -(\tilde{E}^{-1}\tilde{B})(\tilde{E}^{-1}\tilde{B})^T & \equiv \\ AX + XA^T & = & -BB^T. & \end{array} \tag{15}$$

(Note that for numerical stability reasons, E should not be inverted into $\tilde{A}$ and the generalized Lyapunov equation

$$\tilde{A}X\tilde{E}^T + \tilde{E}X\tilde{A}^T = -\tilde{B}\tilde{B}^T$$

should be solved using the sign function variant discussed in [24]. Here, we are interested in the performance and, therefore, just use (15) in order to test the algorithms; the method in [24] can be implemented analogously as the one considered here, this will be the topic of future work.) Table 2 specifies the dimensions of matrices that appear in the two systems.

We first evaluate the performance of our hybrid CPU-GPU implementation of the Newton iteration for the factored solution (Algorithm GECLNC in Section 2.1 combined with the hybrid matrix inversion procedure via GJE). Note that in order to exploit the high performance of the GPU matrix inversion procedure, all computations are performed in single-precision arithmetic.

Table 3 reports the execution times (in seconds) of the different computations and data transfers during the first stages of the Newton iteration: inversion of the matrix A_k^{-1} (via the hybrid variant of GEINV_BLK), matrix product $A_k^{-1}B_k$, transfer of the matrix A_k^{-1} from the GPU to the CPU (Time transfer), total cost of the iteration (Time iter.), and accumulated time (Accum. time). Also, the last column in the table shows the convergence of the iteration, according to the stopping criterion displayed in (10). No compression is applied in this first experiment. These results show that much of the iteration time is spent in the matrix inversion. The matrix product $A_k^{-1}B_k$, on the other hand, represents between 2% and 5% of the time, and increases with the iteration count as the number of columns

#iter. k	Time A_k^{-1}	Time $A_k^{-1}B_k$	Time transfer	Time iter.	Accum. time	Conv. criterion $\|A_k + I_n\|_F/\sqrt{n}$
			`STEEL_I`			
1	1.108	0.035	0.129	1.6000	1.600	1.443e+00
2	1.091	0.027	0.127	1.5720	3.172	3.570e−01
3	1.090	0.032	0.127	1.5740	4.746	1.837e−02
4	1.090	0.045	0.127	1.5870	6.333	6.756e−05
5	1.090	0.071	0.127	1.6140	7.947	3.358e−08
			`FLOW_METER`			
1	7.645	0.105	0.436	9.340	9.340	8.246e+01
2	8.077	0.105	0.437	9.772	19.112	2.119e+00
3	8.063	0.106	0.437	9.757	28.869	4.116e−01
4	8.058	0.106	0.438	9.751	38.620	2.637e−02
5	8.056	0.107	0.437	9.751	48.371	4.503e−03
6	8.056	0.127	0.437	9.770	58.141	5.490e−05
7	8.101	0.174	0.437	9.863	68.004	1.473e−09

Table 3: Performance of the hybrid CPU+GPU implementation of the Newton iteration for the solution of the Lyapunov equation with factored right-hand side.

in B_k doubles with each iteration. Finally, the transfer time is slightly over 10% of the total time for the small problem (`STEEL_I`) but below 5% for the large problem size (`FLOW_MODEL`), which could be expected as the ratio computation/communication is $(2n^3 + O(n^2))/n^2$. The last column in the table illustrates the quadratic convergence of the Newton iteration: 5 and 7 iterations respectively are enough to attain convergence in single-precision arithmetic for examples `STEEL_I` and `FLOW_METER`. Overall, approximately 8 and 68 seconds were required to obtain full-rank factors to the solution of two Lyapunov equations of dimensions, 5,177 and 9,669, in single-precision arithmetic.

Table 4 illustrates the impact of the compression technique in the Newton iteration. (Note that the inversion and transfer times as well as the convergence criterion all depend solely on A_k and, therefore, are not affected by the compression.) For each iteration, the table presents the time required to compute the matrix product $A_k^{-1}B_k$ and compress the resulting matrix $B_{k+1} \leftarrow \frac{1}{\sqrt{2c_k}}\left[B_k,\ A_k^{-1}B_k\right]$ via the QR factorization with column piv-

#iter. k	Time $A_k^{-1}B_k$	Time $\bar{R}_{k+1}$	$m_k \rightarrow \bar{m}_{k+1}$	Time iter.	Accum. time
STEEL_I					
1	0.028	0.007	$14 \rightarrow 14$	1.594	1.594
2	0.028	0.006	$28 \rightarrow 28$	1.577	3.171
3	0.032	0.020	$56 \rightarrow 49$	1.594	4.765
4	0.040	0.045	$98 \rightarrow 62$	1.628	6.393
5	0.044	0.060	$124 \rightarrow 62$	1.647	8.040
FLOW_METER					
1	0.105	0.005	$2 \rightarrow 2$	9.349	9.349
2	0.106	0.002	$4 \rightarrow 4$	9.776	19.125
3	0.106	0.005	$8 \rightarrow 8$	9.764	28.889
4	0.106	0.011	$16 \rightarrow 14$	9.763	38.652
5	0.108	0.020	$28 \rightarrow 20$	9.774	48.426
6	0.113	0.031	$40 \rightarrow 20$	9.789	58.215
7	0.112	0.031	$40 \rightarrow 20$	9.790	68.005

Table 4: Performance of the compression technique hybrid CPU+GPU implementation of the Newton iteration for the solution of the Lyapunov equation with factored right-hand side.

oting (Time $\bar{R}_{k+1}$); the last two columns of the table contain the number of columns of the original and compressed factors ($m_{k+1} \rightarrow \bar{m}_{k+1}$), and the iteration and accumulated times.

The conclusion from this particular experiment is that the compression technique yields an important reduction in the number of columns of the factor B_k for the last iterations. However, this reduction does not show up in the iteration time. For instance, in Example FLOW_METER, the compression at step 6 requires 0.031 seconds, yields a reduction of the number of columns of B_k from 40 to 20, and results in an execution time of 0.112 seconds for the matrix product at step 7. Therefore, the net effect of the use of compression is $0.031+0.112 = 0.143$ seconds. On the other hand, looking up Table 3 we find that computing the matrix product in step 7 requires 0.174 seconds. Thus, even for the last iteration steps, where compression can potentially deliver larger gains, the difference in the execution time is minor. Therefore, in the following experiments we only perform one compression, at the last step of the Newton iteration.

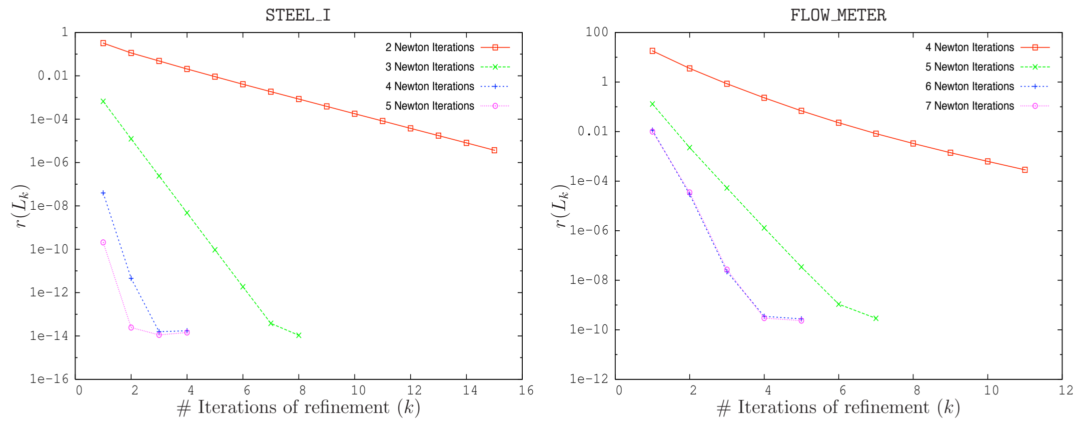

Figure 2: Convergence rate of the iterative refinement procedure.

Figure 2 illustrates the convergence rate of the iterative refinement procedure for these two examples. In particular, the plots report the relative residual $r(L_k) = \|\mathcal{R}(L_k)\|_F / \|L_k L_k^T\|_F$, where L_k is the factor computed after k steps of iterative refinement are applied to the initial approximate factor L_0 computed after $\bar{k}$ =2–7 steps of the Newton iteration. These results show that a rough approximation to the solution ($r(\bar{L}_0) \approx 0.01$) is enough to attain convergence in the iterative refinement. The convergence rate depends on the quality of the approximation and the problem but can be classified as super-linear. Thus, we can expect that a small number of refinement iterations will suffice to attain double-precision accuracy.

Overall, the best global execution time for Example STEEL_I is obtained with 4 steps of the Newton iteration followed by 3 iterations of refinement. This delivers an execution time of 13.089 seconds and a relative residual $r(L_8) = 1.573\text{e-}14$. In Example FLOW_METER, the best combination is obtained with 6/4 steps of Newton iteration/iterative refinement, which require an execution time of 71.667 seconds and yield a relative residual $r(L_8) = 3.484\text{e-}10$.

For comparison, Table 5 reports the execution time and convergence of a implementation of Algorithm GECLNC, with the matrix inversion computed via GJE in the CPU (Algorithm GEING_BLK), and all operations performed in double-precision arithmetic (thus, no refinement is necessary). In Example STEEL_I, the Newton iteration converged in 6 steps, with an execution

#iter. k	Time A_k^{-1}	Time $A_k^{-1}B_k$	Time iter.	Accum. time	Conv. criterion $\|A_k + I_n\|_F/\sqrt{n}$
			`STEEL_I`		
1	8.612	0.127	9.299	9.299	1.443e+00
2	8.569	0.051	9.171	18.470	3.570e−01
3	8.587	0.064	9.202	27.672	1.837e−02
4	8.593	0.085	9.229	36.901	6.755e−05
5	8.582	0.136	9.269	46.170	1.382e−09
6	8.585	0.252	9.388	55.558	5.220e−19
			`FLOW_METER`		
1	55.325	0.255	57.541	57.541	8.251e+01
2	55.799	0.512	58.236	115.787	2.124e+00
3	55.693	0.356	57.977	173.753	4.122e−01
4	55.592	0.167	57.686	231.443	2.644e−02
5	55.721	0.187	57.834	289.270	4.511e−03
6	55.689	0.215	57.831	347.115	5.510e−05
7	55.691	0.295	57.913	405.024	4.932e−09
8	55.688	0.485	58.099	463.122	1.091e−17

Table 5: Performance of the double-precision CPU implementation of the Newton iteration for the solution of the Lyapunov equation with factored right-hand side.

time of 55.558 seconds, and $r(L_6)$ = 4.455e-15. Example `FLOW_METER` required 8 Newton iterations and 463.122 seconds to converge, and attained $r(L_8)$ = 3.422e-09. In summary, compared with this code, the hybrid algorithm attains speed-ups of 4.24/6.46 for Examples `STEEL_I`/`FLOW_METER` while attaining similar numerical accuracy in the relative residual.

4. Conclusions and open questions

We have presented a mixed-precision algorithm, based on the matrix sign function, that allows the efficient solution of Lyapunov equations exploiting the features of current multi-core processors and many-core GPUs. In particular, the huge hardware parallelism of the GPU (in single-precision) is exploited by developing a tailored hybrid implementation of a matrix inversion routine based on GJE, with look-ahead, that overlaps the computations in the CPU and the GPU. This routine is used to off-load to the

GPU the matrix inverses that appear in the Newton iteration for the matrix sign function, by far the most expensive operation in the factored version of this iterative scheme. The Newton iteration produces a full-rank factor with user-prescribed accuracy as it can be stopped with a prescribed tolerance. A second major contribution of this paper is a refinement procedure, which can then be used to cheaply evolve the approximation of the full-rank factor to double-precision accuracy. Overall, we obtain a mixed-precision solver which, compared with an equivalent code that operates only in the CPU and using double-precision arithmetic, attains a speed-up of 4.24 and 6.46 for two Lyapunov equations of order 5,177 and 9,669.

A general conclusion that can be extracted from our work is that current GPUs are an extremely promising approach to solve scientific and engineering applications where dense linear algebra (and specially, dense linear systems) is the key. This paper demonstrates that for a particular application in control theory like model reduction, the use of a standard hardware accelerator can potentially replace a much harder-to-program/operate distributed-memory cluster. Mixed-precision algorithms are expected to play a significant role in future high-performance software and, therefore, iterative refinement procedures like the one introduced in this paper will become important.

A likely concern is that future GPUs may reduce the ratio 8:1 between single and double-precision arithmetic performance (Fermi, NVIDIA next generation GPU already promises a reduction of this factor to 2:1). However, even with a ratio 2:1, it may be preferable to perform a few steps of the Newton iterative scheme, to obtain an initial solution with a few accurate digits (1–2), and then evolve this solution to full, double-precision accuracy with the less expensive refinement procedure.

Acknowledgements

We thank Francisco D. Igual for his technical support and advice in this work. Enrique S. Quintana-Ortí and Alfredo Remón were supported by projects PROMETEO/2009/013 and CICYT TIN2008-06570-C04. This work was partially done while Pablo Ezzatti was visiting the Universidad Jaime I with the support from project PROMETEO/2009/013.

References

[1] A. Antoulas, Approximation of Large-Scale Dynamical Systems, SIAM Publications, Philadelphia, PA, 2005.

[2] P. Benner, V. Mehrmann, D. Sorensen (Eds.), Dimension Reduction of Large-Scale Systems, Lecture Notes in Computational Science and Engineering, Springer-Verlag, Berlin/Heidelberg, Germany, 2005, to appear.

[3] P. Benner, Contributions to the Numerical Solution of Algebraic Riccati Equations and Related Eigenvalue Problems, Logos–Verlag, Berlin, Germany, 1997, *Also:* Dissertation, Fakultät für Mathematik, TU Chemnitz–Zwickau, 1997.

[4] B. Datta, Numerical Methods for Linear Control Systems, Elsevier Academic Press, 2004.

[5] M. Green, D. Limebeer, Linear Robust Control, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[6] IMTEK, `http://www.imtek.de/simulation/benchmark/`, Oberwolfach model reduction benchmark collection.

[7] G. Golub, C. Van Loan, Matrix Computations, 3rd Edition, Johns Hopkins University Press, Baltimore, 1996.

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

[9] S. Hammarling, Numerical solution of the stable, non-negative definite Lyapunov equation, IMA J. Numer. Anal. 2 (1982) 303–323.

[10] R. Granat, B. Kågström, D. Kressner, A novel parallel QR algorithm for hybrid distributed memory HPC systems, Technical report 2009-15, Seminar for applied mathematics, ETH Zurich (April 2009).

[11] R. Granat, B. Kågström, Parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part I: Theory and algorithms, *ACM Transactions on Mathematical Software* (revised January 2009) (July 2007).

[12] R. Granat, B. Kågström, Algorithm XXX: The SCASY software library – parallel solvers for Sylvester-type matrix equations with applications in condition estimation, Part II, *ACM Transactions on Mathematical Software* (revised January 2009) (July 2007).

[13] P. Benner, J. Claver, E. S. Quintana-Ortí, Parallel distributed solvers for large stable generalized Lyapunov equations, Parallel Processing Letters 9 (1) (1999) 147–158.

[14] P. Benner, E. S. Quintana-Ortí, G. Quintana-Ortí, A portable subroutine library for solving linear control problems on distributed memory computers, in: G. Cooperman, E. Jessen, G. Michler (Eds.), Workshop on Wide Area Networks and High Performance Computing, Essen (Germany), September 1998, Lecture Notes in Control and Information, Springer-Verlag, Berlin/Heidelberg, Germany, 1999, pp. 61–88.

[15] P. Benner, E. S. Quintana-Ortí, G. Quintana-Ortí, State-space truncation methods for parallel model reduction of large-scale systems, Parallel Comput. 29 (2003) 1701–1722.

[16] P. Benner, E. S. Quintana-Ortí, G. Quintana-Ortí, Solving linear-quadratic optimal control problems on parallel computers, Optimization Methods & Software 23 (6) (2008) 879–909.

[17] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, Evaluation and tuning of the level 3 CUBLAS for graphics processors, in: Proceedings of the 10th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2008, 2008, pp. CD–ROM.

[18] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, Solving dense linear systems on graphics processors, in: E. Luque, T. Margalef, D. Benítez (Eds.), Proceedings of the 14th international Euro-Par conference on Parallel Processing, Lecture Notes in Computer Science, 5168, Springer, 2008, pp. 739–748.

[19] V. Volkov, J. Demmel, LU, QR and Cholesky factorizations using vector capabilities of GPUs, Tech. Rep. UCB/EECS-2008-49, EECS Department, University of California, Berkeley (May 2008).

URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-20%08-49.html

[20] P. Bientinesi, F. D. Igual, D. Kressner, E. S. Quintana-Ortí, Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures, in: Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics – PPAM'09, Lecture Notes in Computer Science, Springer, to appear.

[21] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, E. S. Quintana-Ortí, G. Quintana-Ortí, Exploiting the capabilities of modern GPUs for dense matrix computations, Concurrency and Computation: Practice and Experience 21 (2009) 2457–2477.

[22] G. Golub, C. V. Loan, Matrix Computations, 3rd Edition, The Johns Hopkins University Press, Baltimore, 1996.

[23] R. Byers, Solving the algebraic Riccati equation with the matrix sign function, Linear Algebra Appl. 85 (1987) 267–279.

[24] P. Benner, E. S. Quintana-Ortí, Solving stable generalized Lyapunov equations with the matrix sign function, Numer. Algorithms 20 (1) (1999) 75–100.

[25] J. Roberts, Linear model reduction and solution of the algebraic Riccati equation by use of the sign function, Internat. J. Control 32 (1980) 677–687, (Reprint of Technical Report No. TR-13, CUED/B-Control, Cambridge University, Engineering Department, 1971).

[26] V. Sima, P. Benner, Experimental evaluation of the new SLICOT solvers for linear matrix equations based on the matrix sign function, in: Proc. of 2008 IEEE Multi-conference on Systems and Control, 9th IEEE Int. Symp. on Computer-Aided Systems Design (CACSD), 2008, pp. 601–606.

[27] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, D. Sorensen, LAPACK Users' Guide, SIAM, Philadelphia, 1992.

[28] P. Benner, P. Ezzatti, E. S. Quintana-Ortí, A. Remón, Using hybrid CPU-GPU platforms to accelerate the computation of the matrix sign

function, in: Proc. 7th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks – HeteroPar'09, Lecture Notes in Computer Science, 2009, to appear.

[29] P. Strazdins, A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization, Tech. Rep. TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia (1998).

[30] U. Baur, P. Benner, Factorized solution of the Lyapunov equation by using the hierarchical matrix arithmetic, Proc. Appl. Math. Mech. 4 (1) (2004) 658–659.

[31] L. Grasedyck, W. Hackbusch, B. Khoromskij, Solution of large scale algebraic matrix Riccati equations by use of hierarchical matrices, Computing 70 (2003) 121–165.

[32] N. J. Higham, Accuracy and Stability of Numerical Algorithms, 2nd Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002.

[33] Texas Advanced Computing Center, `http://www.tacc.utexas.edu/~kgoto/`.

[34] Intel Corporation., `http://www.intel.com/`.

[35] Nvidia Corporation, `http://www.nvidia.com/cuda/`.

[36] F. Tröltzsch, A. Unger, Fast solution of optimal control problems in the selective cooling of steel, Z. Angew. Math. Mech. 81 (2001) 447–456.

[37] H. Ernst, High-resolution thermal measurements in fluids, Ph.D. thesis, University of Freiburg, Germany (2001).

Research Reports

No.	Authors/Title
09-40	*P. Benner, P. Ezzatti, D. Kressner, E.S. Quintana-Ortí, A. Remón* A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms
09-39	*V. Wheatley, P. Huguenot, H. Kumar* On the role of Riemann solvers in discontinuous Galerkin methods for magnetohydrodynamics
09-38	*E. Kokiopoulou, D. Kressner, N. Paragios, P. Frossard* Globally optimal volume registration using DC programming
09-37	*F.G. Fuchs, A.D. McMurray, S. Mishra, N.H. Risebrom, K. Waagan* Approximate Riemann solvers and stable high-order finite volume schemes for multi-dimensional ideal MHD
09-36	*Ph. LeFloch, S. Mishra* Kinetic functions in magnetohydrodynamics with resistivity and hall effects
09-35	*U.S. Fjordholm, S. Mishra* Vorticity preserving finite volume schemes for the shallow water equations
09-34	*S. Mishra, E. Tadmor* Potential based constraint preserving genuinely multi-dimensional schemes for systems of conservation laws
09-33	*S. Mishra, E. Tadmor* Constraint preserving schemes using potential-based fluxes. III. Genuinely multi-dimensional central schemes for for MHD equations
09-32	*S. Mishra, E. Tadmor* Constraint preserving schemes using potential-based fluxes. II. Genuinely multi-dimensional central schemes for systems of conservation laws
09-31	*S. Mishra, E. Tadmor* Constraint preserving schemes using potential-based fluxes. I. Multidimensional transport equations
09-30	*D. Braess, S. Sauter, C. Schwab* On the justification of plate models
09-29	*D. Schötzau, C. Schwab, T. Wihler* *hp*-dGFEM for second-order elliptic problems in polyhedra. II: Exponential convergence