Unleashing CPU-GPU Acceleration for Control Theory Applications

Peter Benner[1], Pablo Ezzatti[2], Enrique S. Quintana-Ortí[3], and Alfredo Remón[3]

[1] Max Planck Institute for Dynamics of Complex Technical Systems, D-39106 Magdeburg, Germany
benner@mpi-magdeburg.mpg.de
[2] Instituto de Computación, Universidad de la República, 11.300-Montevideo, Uruguay
pezzatti@fing.edu.uy
[3] Dpto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain
{quintana,remon}@icc.uji.es

Abstract. In this paper we review the effect of two high-performance techniques for the solution of matrix equations arising in control theory applications on CPU-GPU platforms, in particular advanced optimization via look-ahead and iterative refinement. Our experimental evaluation on the last GPU-generation from NVIDIA, "Kepler", shows the slight advantage of matrix inversion via Gauss-Jordan elimination, when combined with look-ahead, over the traditional LU-based procedure, as well as the clear benefits of using mixed precision and iterative refinement for the solution of Lyapunov equations.

Keywords: Control theory, matrix equations, matrix inversion, multicore processors, graphics processors.

1 Introduction

Consider a dynamic linear time-invariant (LTI) system represented, in the state-space model, as

$$\begin{array}{ll} \dot{x}(t) = Fx(t) + Bu(t), & t > 0, \quad x(0) = x^0, \\ y(t) = Cx(t) + Du(t), & t \geq 0, \end{array} \tag{1}$$

where $x(t) \in \mathbb{R}^n, u(t) \in \mathbb{R}^m$ and $y(t) \in \mathbb{R}^m$ contain, respectively, the states, inputs and outputs of the system, while $x^0 \in \mathbb{R}^n$ stands for its initial state. Here, $F \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, $C \in \mathbb{R}^{p \times n}$, $D \in \mathbb{R}^{p \times m}$, n is referred to as the order of the system and, usually, the number of inputs and outputs satisfy $m, p \ll n$. Two important control theory applications are *model order reduction* (MOR) and *linear-quadratic optimal control* (LQOC). In the first one, the goal is to find an alternative dynamic LTI system, of order $r \ll n$, which can accurately replace the original system (1) in subsequent operations [2]. On the other hand, the objective

I. Caragiannis et al. (Eds.): Euro-Par 2012 Workshops, LNCS 7640, pp. 102–111, 2013.

of LQOC is to determine an "optimal" feedback control law $u(t) = -Kx(t),\ t \geq 0$, with $K \in \mathbb{R}^{m \times n}$, that stabilizes (1) (i.e., so that all the eigenvalues of $F - BK$ have negative imaginary part) [14].

Large-scale dynamic LTI systems ($n \geq 5,000$) often arise when modeling controlled physical processes by means of partial differential equations [1,2,9]. Reliable numerical methods for MOR and LQOC problems require the solution of certain linear and quadratic matrix equations featuring a high computational cost. For instance, solving the Lyapunov equations for the controllability/observability Gramians associated with (1), via the matrix sign function, roughly requires $2n^3$ floating-point arithmetic operations (or flops) per iteration, with the number of iterations required for convergence varying between 3–4 to a few dozens [15]. Given the theoretical peak performance of current CPU cores (4 double-precision flops/cycle), we can thus estimate that, under perfect conditions (i.e., operating at peak performance for the full execution of the solver), performing one iteration of the matrix sign function on a single CPU core, for an equation of order $n = 10^4$, can cost slightly more than 4 minutes; if the order of the equation grows to $n = 10^5$, the execution time of a single iteration is longer than 69 hours!

In the past [7,8], we have shown how the use of message-passing Lyapunov solvers based on the matrix sign function (and kernels from the distributed-memory linear algebra library ScaLAPACK [10]) provides an appropriate means to solve MOR and LQOC problems of moderate scale ($n \approx$ 5,000–10,000) on a 32-node cluster. Following the trend of adopting graphics processors (GPUs) as hardware accelerators for compute-intensive applications, more recently we have developed hybrid CPU-GPU codes for the solution of these control applications. For instance, we have evaluated the performance of a basic building kernel like the matrix inverse on a platform consisting of a general-purpose multicore from Intel and a "Fermi" GPU from NVIDIA [5].

In this paper we review two advanced optimization techniques for the solution of Lyapunov equations via the matrix sign function on CPU-GPU platforms: 1) the use of look-ahead in the framework of computing the matrix inverse; and 2) the combined use of mixed precision and iterative refinement (MPIR) to accelerate the solution of Lyapunov matrix equations. While a significant part of the theoretical aspects underlying this work has already been exposed in previous work (see, e.g., [5,6]), the principal motivation for revisiting these techniques is the latest evolution of GPU architectures, specifically, the new "Kepler" GPU from NVIDIA, featuring an important increase in the number of cores w.r.t. the previous generation (Fermi), and also a very different ratio of single-to-double precision arithmetic performance.

The rest of the paper is organized as follows. In Section 2 we briefly review the use of the matrix sign function to solve the matrix equations arising in MOR and LQOC problems. In Section 3 we expose how to efficiently combine look-ahead with several basic optimization techniques for CPU-GPU platforms, in the context of matrix inversion. There, we also illustrate the impact of these techniques on the performance of our hybrid CPU-GPU algorithms on a platform equipped

with a Kepler. In Section 4 we revisit a MPIR procedure in combination with the sign function-based solver for the Lyapunov equation, and experimentally evaluate its performance on the target platform. Finally, we complete the paper with some remarks in Section 5.

2 Solving Matrix Equations via the Matrix Sign Function

Balanced truncation (BT) is an efficient absolute-error method for MOR of large-scale LTI systems [2]. The crucial operation when applying BT to (1) is the solution of the dual Lyapunov equations

$$FW_c + W_cF^T + BB^T = 0, \quad F^TW_o + W_oF + C^TC = 0, \tag{2}$$

for the Cholesky (or, alternatively, low-rank) factors of the symmetric positive semi-definite (s.p.d.) Gramians $W_c, W_o \in \mathbb{R}^{n\times n}$. On the other hand, given a pair of weight matrices $R \in \mathbb{R}^{m\times m}$ and $Q \in \mathbb{R}^{p\times p}$, with R s.p.d. and Q symmetric, under certain conditions the optimal control law for the LQOC problem is given by $u(t) = -Kx(t) = -R^{-1}B^TXx(t)$, with $X \in \mathbb{R}^{n\times n}$ being the s.p.d. solution of the algebraic Riccati equation (ARE):

$$F^TX + XF - XBR^{-1}B^TX + C^TQC = 0. \tag{3}$$

Given a matrix $A \in \mathbb{R}^{q\times q}$, the Newton iteration for the matrix sign function is defined as

$$A_0 := A, \quad A_{j+1} := \frac{1}{2}(A_j + A_j^{-1}), \quad j = 0, 1, \ldots. \tag{4}$$

Provided A has no eigenvalues on the imaginary axis, $\lim_{j\to\infty} A_j = \operatorname{sign}(A)$ with an asymptotically quadratic convergence rate. Both the solution of the Lyapunov equations in (2) and the ARE in (3) can be obtained via specialized variants of (4). In both cases, the key operation from the point of view of the cost is the computation of the inverse of an $n \times n$ (Lyapunov) or $2n \times 2n$ (ARE) matrix; see [15] for details. Computing the inverse of a matrix costs $2n^3$ flops, if the matrix has no special structure, or just n^3 flops, in case symmetry can be exploited. In general, the inverse of a sparse, banded or tridiagonal matrix is dense and, therefore, these special structures cannot be leveraged to reduce the cost of computing the matrix inverse during the sign function iteration.

3 Efficient Matrix Inversion on CPU-GPU Platforms

While there exist different approaches for the inversion of (general, symmetric and s.p.d.) matrices, in past work [5] we have shown the superior performance of Gauss-Jordan elimination (GJE) over the conventional LU-based matrix inversion when the target is a heterogeneous platform that combines a traditional CPU with a GPU. The reason for the advantage of matrix inversion via GJE is

twofold. First, the procedure performs a constant number of flops per step, facilitating a balanced distribution of the computation between the CPU cores and the GPU. Second, GJE is richer in large matrix-matrix multiplies, an operation that delivers a high FLOPS (flops/sec.) ratio on both CPUs and GPUs.

Figure 1 shows a blocked procedure for the inversion of a general matrix via GJE (right) and the unblocked variant upon which it is built (left). For simplicity, row permutations are not included in our following discussion, though in practice all our GJE-inversion algorithms employ partial pivoting to ensure practical stability of the procedure.

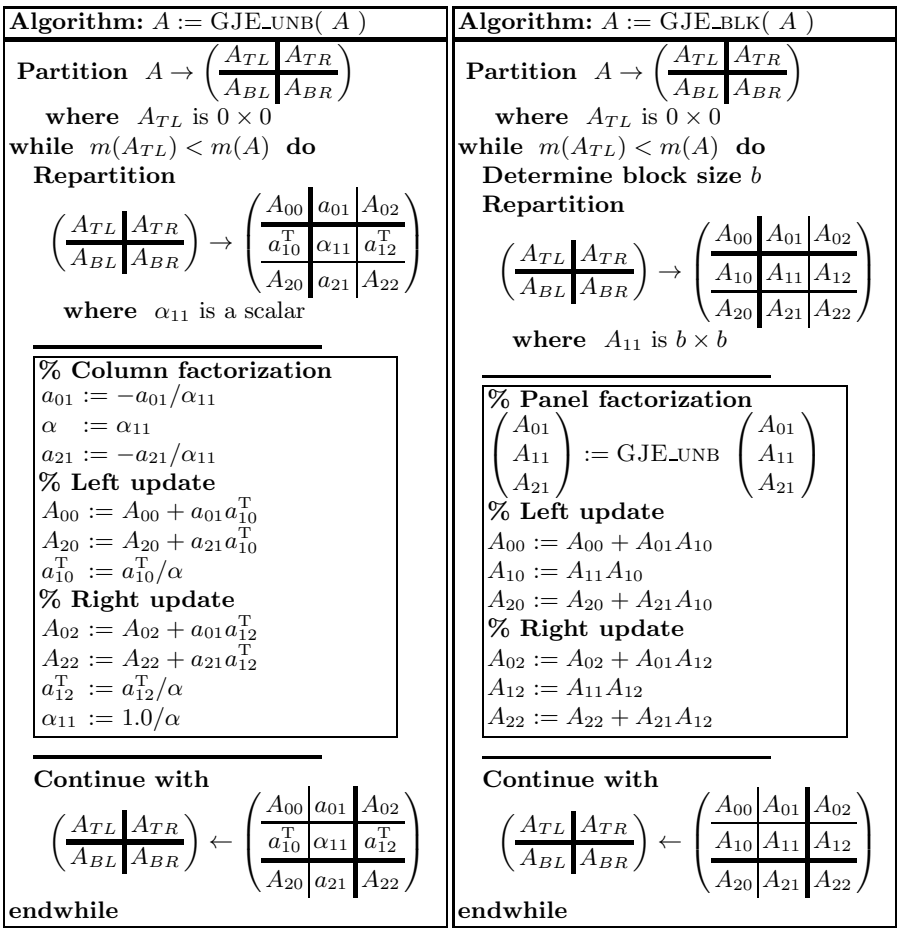

Fig. 1. Unblocked and blocked algorithms (left and right, respectively) for the inversion of a general matrix via GJE. Here, $m(\cdot)$ returns the number of rows of its argument.

3.1 Optimization

Look-ahead is a high performance technique to overcome performance bottlenecks due to the execution of serial phases during the computation of dense linear algebra operations [17]. This strategy is applied in tuned linear algebra libraries (Intel MKL, AMD ACML, the HPL Linpack benchmark, etc.) to overlap the factorization of a block panel in the LU and QR factorizations with the update of the remaining parts of the matrix. The resulting procedure yields superior performance at the cost of higher programming complexity, especially if several levels of look-ahead are simultaneously applied. Alternatively, a dynamic variant of look-ahead can be easily attained by employing a runtime like, e.g., SMPSs [16,3] to schedule a dense linear algebra operation decomposed into a number of tasks with dependencies among them.

Consider the specific case of matrix inversion via the blocked GJE-based matrix inversion procedure in Figure 1 (right), and the partitioning

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) = \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(A_0 \left| A_1 \right| A_2^L \left| A_2^R \right.\right),$$

set at the beginning of the first iteration of the loop, where A_{TL}, A_{00} are both 0×0, correspondingly A_0 contains no columns, and A_1 has b columns. Assume that A_2^L also contains b columns. The idea of look-ahead is, during this first iteration, to factorize the panel A_1 and, immediately after, perform the corresponding update of A_2^L; next, the updates of A_0 and A_2^R are overlapped with the factorization of A_2^L during this initial iteration. One can then apply the same idea, during subsequent iterations $k = 2, 3, \ldots$, to overlap the factorization of the $(k+1)$-th block panel with the updates corresponding to the k-th iteration.

The appealing property of the GJE-based matrix inversion is that the amount of flops per iteration remains constant, easily accommodating look-ahead and enhancing its performance advantage compared with factorizations that operate on a decreasing number of data (e.g., LU, QR or Cholesky). This property can be also leveraged to obtain a balanced distribution of the computations between the CPU and the GPU in a heterogeneous platform. In particular, due to the complexity of the panel factorization (especially when partial pivoting is applied), it is more convenient to perform this operation on the CPU. On the other hand, the (left and right) updates can be off-loaded to the GPU except, possibly, for a small panel that can be computed on the CPU.

Consider now the data transfers required in this hybrid CPU-GPU matrix inversion procedure. We can initially transfer the full matrix A from the CPU (memory), via the PCI-e, to the GPU (memory). Given that this requires $n \times n$ memory operations (memops) for $2n^3$ flops, the communication cost is negligible for large n.

Consider next the data transfers that occur during the inversion procedure. At each iteration of the algorithm, a panel of b columns of A has to be transferred from GPU to CPU, factorized there, and the result has to be sent back to the GPU for the application of the corresponding updates. Thus, this requires

$2(n \times b)$ memops per iteration, which can be amortized with the $2n(n-b)b$ flops corresponding to the update provided b is chosen to be large enough (in practice, for optimal performance, $b \approx 700$ or larger).

Two basic optimization techniques can be applied to further enhance the performance of the hybrid CPU-GPU implementation. First, given that the block size b is moderately large, higher efficiency can be attained from the execution of the panel factorization on a multicore CPU if this is performed by using the blocked algorithmic procedure, with a block size $\hat{b} < b$ (and, usually, with values of 16 or 32 for $\hat{b}$). Second, the matrix-matrix products to be performed in the GPU as part of the update can be combined into operations of larger granularity, improving the FLOPS ratio in these architectures. In particular, consider for example the (left) update of the three blocks that compose A_0. These operations can then be combined into (two matrix manipulation operations and) a single matrix-matrix product as follows:

$$\hat{A} := A_{10}, \quad A_{10} := 0$$
$$\begin{pmatrix} A_{00} \\ A_{10} \\ A_{20} \end{pmatrix} := \begin{pmatrix} A_{00} \\ A_{10} \\ A_{20} \end{pmatrix} + \begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \end{pmatrix} \hat{A}$$

3.2 Experimental Evaluation

The routines for matrix inversion have been evaluated on a platform equipped with an Intel Xeon E5640 at 2.67GHz and a NVIDIA GeForce GTX 680 "Kepler". Intel MKL (version 10.3.4), MAGMA (version 1.2.1) and NVIDIA CUBLAS (version 4.2.9) provided high-performance implementations of the necessary linear algebra kernels. We evaluate the performance of these routines in terms of GFLOPS (10^9 flops/sec.).

Figure 2 reports the performance obtained for the computation of the matrix inverse via three different routines and single-precision (SP) arithmetic. The black dashed line corresponds to the execution of LAPACK routines `sgetrf` + `sgetri` on the multicore processor. The red line shows the results obtained by the best GPU-based variant that computes the inverse using the LU factorization. This variant, named LU+GPU, employs a routine from the library MAGMA [13] to compute the LU factorization, an optimized routine developed by AICES-RWTH to obtain the inverse of the triangular matrix, and our *ad-hoc* implementation to solve the triangular system. (Previous results have demonstrated this as being the combination that delivers highest performance for an LU-based matrix inversion on GPUs [5].) The blue line shows the performance of the GJE-based routine on the GPU. The experimental results demonstrate the superior scalability of this last implementation. Although the LU-based implementation delivers a higher GFLOPS rate than the GJE-based counterpart for the inversion of matrices of dimension up to 5,000, for larger matrices, the GJE implementation offers the best results.

Figure 3 evaluates the same three routines using double-precision (DP) arithmetic in this case, showing that the GJE variant is consistently the best option

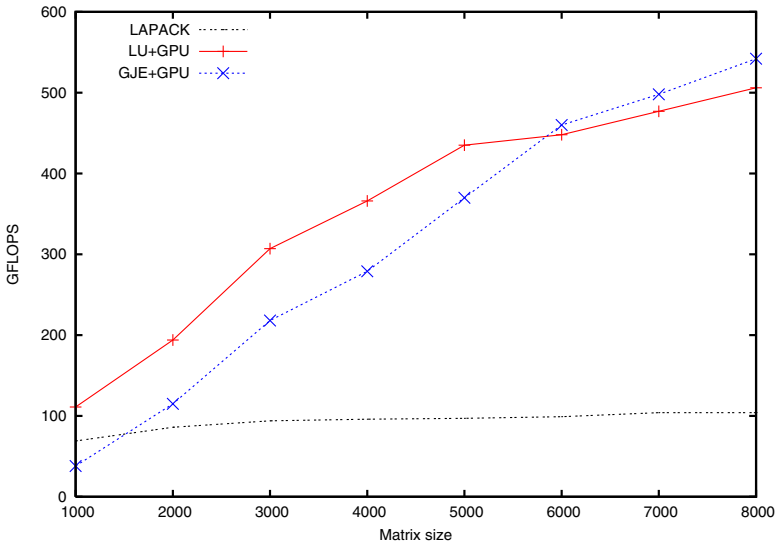

Fig. 2. Performance obtained for the inversion of matrices using SP arithmetic

regardless of the matrix dimension. The results in these experiments also illustrate the performance drop incurred by the introduction of DP, a factor between $2\times$ and $5\times$, motivating the next section.

4 MPIR for the Lyapunov Equation

Iterative refinement is a well-known technique to improve an approximate solution to a linear system of equations [11]. This technique has received renewed interest due to the performance gap between SP and DP in recent hardware accelerators, in particular, GPUs [4].

4.1 Refinement Procedure

Consider we have computed an initial, SP low-rank factor for the controllability Gramian of the Lyapunov $FW_c + W_cF^T + BB^T = 0$ via, e.g., the matrix sign function method. Hereafter, we will refer to this low-rank factor as L_0^S (so that $Y^S := L_0^S(L_0^S)^T$ is a SP approximation to the controllability Gramian W_c); and denote the corresponding computation/procedure as $L_0^S := \texttt{ApproxLyap}(F, B)$. (Note that, in principle, this initial factor could have been computed using some other numerical method different from the matrix sign function.) In [6], the following iterative procedure is introduced to refine this SP factor to the desired precision:

$$\mathcal{R}(L_k) := FL_kL_k^T + L_kL_k^TF^T + BB^T, \tag{5}$$

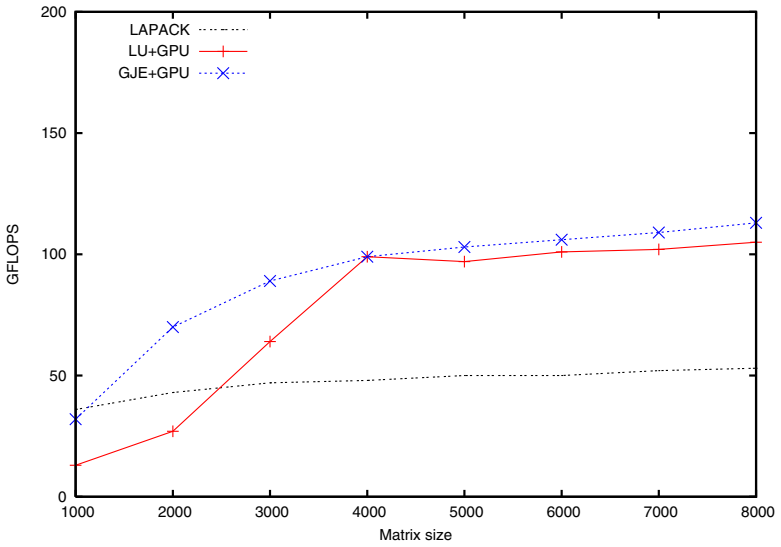

Fig. 3. Performance obtained for the inversion of matrices using DP arithmetic

$$\text{Decompose } \mathcal{R}(L_k) \rightarrow B_+ B_+^T - B_- B_-^T, \tag{6}$$

$$L_+ := \texttt{ApproxLyap}(F, B_+), \tag{7}$$

$$L_- := \texttt{ApproxLyap}(F, B_-), \tag{8}$$

$$Y_{k+1} := L_k L_k^T + L_+ L_+^T - L_- L_-^T, \tag{9}$$

$$\text{Decompose } Y_{k+1} \rightarrow L_{k+1}^T L_{k+1}^T - \hat{L}_- \hat{L}_-^T, \tag{10}$$

with $k = 0, 1, \ldots$. In practice, (5) and (9) are never explicitly constructed. Also, (7) and (8) are computed using SP arithmetic; if the solution procedure is based on the matrix sign function, its computational cost becomes negligible in case the inverses that appear during the iteration are calculated once (e.g., during the initial solution $L_0^S := \texttt{ApproxLyap}(F, B)$), and saved for reuse during the refinement steps. Finally, (6) and (10) require DP arithmetic but are cheap to compute. For further details, see [6].

4.2 Experimental Evaluation

In this section we evaluate the MPIR approach in the platform described in section 3.2. For the experiments we employ the benchmark STEEL from the *Oberwolfach Model Reduction Benchmark Collection* [12]. This LTI system arises in a manufacturing method for steel profiles, where the objective is to design a control to assure the quality of the steel profile obtaining moderate temperature gradients while the rail is cooled down. In particular, the instance of the STEEL problem employed in this work has $n = 5,177$ state variables, $m = 7$ inputs and $p = 6$ outputs.

Table 1. Results obtained for the STEEL case using the MPIR technique

#Iter. Sign	#Iter. ref.	Residual	Time (s)
4	1	2.1e-10	6.7
4	2	6.5e-12	7.7
4	3	1.3e-13	8.9
4	4	2.9e-15	10.7

Table 1 presents the results obtained to solve the STEEL problem using the MPIR technique. The first and second columns show, respectively, the number of iterations of the sign function (in SP) and refinement iterations. Column 3 reports the residual $\|\mathcal{R}(L_{\bar{k}+k})\|_F / \|L_{\bar{k}+k} L_{\bar{k}+k}^T\|_F$, where $L_{\bar{k}+k}$ denotes the factor computed when k steps of iterative refinement are applied to the initial approximate factor computed after $\bar{k}$ steps of the Newton iteration for the sign function; and column 4 corresponds to the execution time of the complete solver (sign function+MPIR) in seconds. The results demonstrate the moderate execution time added by MPIR. An accurate solution is obtained with 4 sign function iterations followed by 4 refinement steps, yielding an execution time of 10.7 seconds. A solution of similar accuracy using a DP implementation of the Newton iteration for the matrix sign function requires over 20 seconds.

5 Concluding Remarks

We have evaluated the use of two optimization techniques for control theory problems, namely, look-ahead and MPIR, on the new Kepler architecture. Look-ahead facilitates the concurrent execution of operations, allowing the concurrent use of many computational units, e.g., during matrix inversion via GJE on CPU-GPU platforms. The MPIR technique delivers DP accuracy while performing most of the computations in SP arithmetic. This approach is specially appealing on current GPUs, where SP performance is between 4 and 5× faster than DP. Experimental results show convenience of both techniques, demonstrating clear performance gains for the solution of control theory problems on heterogeneous platforms equipped with a GPU.

Acknowledgements. The researchers from the Universidad Jaume I (UJI) were supported by project TIN2011-23283 and FEDER.

References

1. Abels, J., Benner, P.: DAREX – a collection of benchmark examples for discrete-time algebraic Riccati equations (version 2.0). SLICOT Working Note 1999-15 (November 1999), `http://www.slicot.org`
2. Antoulas, A.: Approximation of Large-Scale Dynamical Systems. SIAM Publications, Philadelphia (2005)

3. Badia, R.M., Herrero, J.R., Labarta, J., Pérez, J.M., Quintana-Ortí, E.S., Quintana-Ortí, G.: Parallelizing dense and banded linear algebra libraries using SMPSs. Concurrency and Computation: Practice and Experience 21(18), 2438–2456 (2009)
4. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S., Quintana-Ortí, G.: Exploiting the capabilities of modern GPUs for dense matrix computations. Concurrency and Computation: Practice and Experience 21, 2457–2477 (2009)
5. Benner, P., Ezzatti, P., Quintana-Ortí, E., Remón, A.: Matrix inversion on CPU-GPU platforms with application in control theory. Concurrency and Computation: Practice and Experience (to appear, 2012)
6. Benner, P., Ezzatti, P., Kressner, D., Quintana-Ortí, E., Remón, A.: A mixed-precision algorithm for the solution of Lyapunov equations on hybrid CPU-GPU platforms. Parallel Computing 37(8), 439–450 (2011)
7. Benner, P., Quintana-Ortí, E., Quintana-Ortí, G.: State-space truncation methods for parallel model reduction of large-scale systems. Parallel Computing 29, 1701–1722 (2003)
8. Benner, P., Quintana-Ortí, E., Quintana-Ortí, G.: Solving linear-quadratic optimal control problems on parallel computers. Optimization Methods Software 23(6), 879–909 (2008)
9. Chahlaoui, Y., Van Dooren, P.: A collection of benchmark examples for model reduction of linear time invariant dynamical systems. SLICOT Working Note 2002–2 (February 2002), `http://www.slicot.org`
10. Choi, J., Dongarra, J.J., Pozo, R., Walker, D.W.: ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In: Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation, pp. 120–127. IEEE Computer Society Press (1992)
11. Higham, N.J.: Accuracy and Stability of Numerical Algorithms, 2nd edn. Society for Industrial and Applied Mathematics, Philadelphia (2002)
12. IMTEK, Oberwolfach model reduction benchmark collection, `http://www.imtek.de/simulation/benchmark/`
13. MAGMA project home page, `http://icl.cs.utk.edu/magma/`
14. Mehrmann, V.: The Autonomous Linear Quadratic Control Problem, Theory and Numerical Solution. LNCIS, vol. 163. Springer, Heidelberg (1991)
15. Roberts, J.: Linear model reduction and solution of the algebraic Riccati equation by use of the sign function. Internat. J. Control 32, 677–687 (1980); Reprint of Technical Report No. TR-13, CUED/B-Control, Cambridge University, Engineering Department (1971)
16. SMP superscalar project home page, `http://www.bsc.es/plantillaG.php?cat_id=385`
17. Strazdins, P.: A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Technical Report TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia (1998)