Peter Benner Pablo Ezzatti Enrique S. Quintana-Ortí
Alfredo Remón

Matrix Inversion on CPU-GPU Platforms with Applications in Control Theory

Max Planck Institute Magdeburg Preprints

MPIMD/12-02 **February 1, 2012**

Impressum:

Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg

Publisher:
Max Planck Institute for Dynamics of Complex Technical Systems

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

www.mpi-magdeburg.mpg.de/preprints

Matrix Inversion on CPU-GPU Platforms with Applications in Control Theory *

Peter Benner† Pablo Ezzatti‡
Enrique S. Quintana-Ortí§ Alfredo Remón¶

February 1, 2012

Abstract

In this paper we tackle the inversion of large-scale dense matrices via conventional matrix factorizations (LU, Cholesky, LDL^T) and the Gauss-Jordan method on hybrid platforms consisting of a multi-core CPU and a many-core graphics processor (GPU). Specifically, we introduce the different matrix inversion algorithms using a unified framework based on the notation from the FLAME project; we develop hybrid implementations for those matrix operations underlying the algorithms, alternative to those in existing libraries for single-GPU systems; and we perform an extensive experimental study on a platform equipped with state-of-the-art general-purpose architectures from Intel and a "Fermi" GPU from NVIDIA that exposes the efficiency of the different inversion approaches. Our study and experimental results show the simplicity and performance advantage of the GJE-based inversion methods, and the difficulties associated with the symmetric indefinite case.

Keywords: Matrix inversion, matrix equations, dense linear algebra, control theory, high performance computing, graphics processors.

*The researchers from the UJI were supported by project TIN2011-23283 and FEDER.

†Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, Germany. benner@mpi-magdeburg.mpg.de.

‡Centro de Cálculo-Instituto de Computación, Universidad de la República, Montevideo, Uruguay. pezzatti@fing.edu.uy.

§Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12.071–Castellón, Spain. quintana@icc.uji.es.

¶Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaume I (UJI), 12.071–Castellón, Spain. remon@icc.uji.es.

1 Introduction

Explicit matrix inversion is required in a few selected applications. One important case is the computation of the *sign function* of a matrix $A \in \mathbb{R}^{s \times s}$ via the Newton iteration [18]:

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

Provided A has no eigenvalues on the imaginary axis, the sequence $\{A_j\}_{j=0}^{\infty}$ converges to sign (A), at an asymptotically quadratic rate. Diverse variants of (1), all requiring the explicit inversion of a matrix at each iteration, can be used to efficiently solve several types of matrix equations arising in numerically robust methods for model reduction and linear-quadratic optimal control (LQOC) [6, 7]. Large-scale instances of these problems involving dense matrices arise, e.g., in VLSI circuit simulation [1] and, when tackled via the matrix sign function, require the inversion of a dense matrix of dimension $s \rightarrow O(10,000 - 100,000)$.

Efficient and numerically stable methods for the inversion of a dense matrix A first compute some type of factorization of the matrix to then obtain the inverse from the resulting factors. Depending on the properties of A, the initial stage is usually performed via the LU factorization with partial pivoting (if A has no special properties), the Cholesky decomposition (when A is symmetric positive definite, s.p.d.), or the LDL^T factorization (when A is symmetric indefinite). If A is of size s, these three types of decompositions as well as the computation of the inverse from the triangular factors require $O(s^3)$ flops (floating-point arithmetic operations). Alternatively, in the former two cases one can employ Gauss-Jordan elimination (GJE), also at a cost of $O(s^3)$ flops. Thus, these methods clearly ask for high performance computing techniques when s is large.

Research in the framework of the MAGMA and FLAME projects has demonstrated the potential of graphics processors (GPUs) to accelerate the computation of dense linear algebra operations [16, 20]. These two projects, together with the commercial development CULA [10], provide hybrid CPU-GPU kernels for the LU factorization and the Cholesky decomposition, among others. In [3, 11, 4, 5] we have reported the results from our own effort towards the computation of dense linear algebra operations, with special focus on matrix inversion and its application to the solution of Lyapunov matrix equations and algebraic Riccati equations (AREs) arising in control theory. In this paper we extend those results as follows:

- We provide a unified framework, based on the FLAME notation [14, 8], for the presentation of conventional matrix inversion algorithms via the LU and Cholesky factorizations (for general and s.p.d. matrices, respectively), and alternative approaches based on the GJE (for both cases).

- We include new multi-core and hybrid algorithms for the inversion of symmetric indefinite matrices via the LDL^T factorization, a case not tackled in previous work, accommodating them into the same framework/notation.

- We perform an experimental study of the performance and overheads of our implementations of the inversion algorithms, comparing them with those available in the MAGMA and CULA libraries, using a state-of-the-art platform equipped with Intel Xeon-based multi-core processors and a NVIDIA "Fermi" GPU.

The rest of the paper is structured as follows. In the next subsection we introduce an algorithmic skeleton that will allow a unified presentation of the matrix inversion algorithms. In Section 2 we briefly review how to use the matrix sign function to solve the Lyapunov equation and the ARE, and the application of these matrix equations in model reduction and LQOC. The three algorithmic approaches and the hybrid CPU-GPU implementations for the inversion of general, s.p.d., and symmetric indefinite matrices are presented, respectively, in Sections 3, 4, and 5. In Section 6 we describe several optimization techniques, applicable to most hybrid algorithms. Experimental results on a hybrid CPU-GPU platform are reported in Section 7, and a few remarks close the paper in Section 8.

1.1 Dense linear algebra algorithms

High performance algorithms for dense linear algebra operations organize the computations by blocks, to hide the memory latency by amortizing the cost of data transfers between the main memory and the floating-point units with a large number of flops. Figure 1 shows two algorithmic frameworks (skeletons), employing the FLAME notation, that illustrate the usual organization of blocked dense linear algebra algorithms. In the figure, $m(\cdot)$ stands for the number of rows of a matrix. The partitioning (**Partition**) before the **while** loop sets $A_{BR} := A$ (left) or $A_{TL} := A$ (right). The repartitionings inside the loop body (**Repartition** and **Continue with**) and the thick lines capture how the algorithms progress through the operands. Thus, the algorithm on the left processes the matrix from the top-left corner to the bottom-right one (TL→BR), while the algorithm on the right proceeds in the opposite direction. At each iteration, both algorithms identify a $b \times b$ diagonal block A_{11}, with b usually known as the (algorithmic) block size. Examples of the first type of algorithms (TL→BR2×2) include the three key factorizations for the solution of dense linear systems: LU, Cholesky and LDL^T. Examples of the second (BL→TL2×2) include the solution of an upper triangular linear system [13].

In the following sections, when presenting an algorithm for a dense linear algebra operation, we will indicate whether it proceeds TL→BR2×2 or BL→TL2×2, and specify the computations that are performed during the "current" iteration (to be included in the box marked as "Update" in one of the algorithms in Figure 1). In our notation TRIU$(\cdot)$/TRIL$(\cdot)$ stands for the upper/lower triangular part of its argument and TRILS$(\cdot)$ for its strictly lower triangular part; TRILU$(\cdot)$ is analogous to TRILS$(\cdot)$ with the diagonal entries of the matrix replaced by ones.

For simplicity, we will not include pivoting in the algorithmic descriptions, though this is crucial for the numerical stability of the inversion of general as well as symmetric indefinite matrices. Our hybrid implementations for the inversion of general matrices employ partial pivoting. Our inversion algorithms for symmetric indefinite matrices

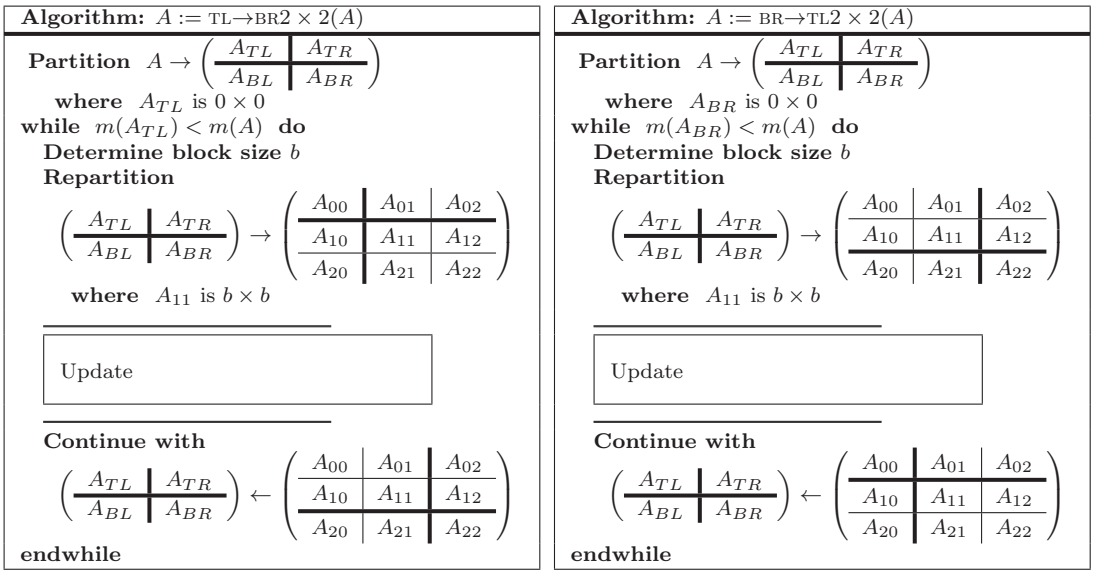

Figure 1: Blocked algorithmic skeletons for dense linear algebra operations.

incorporate pivoting akin that of the LDL^T factorization.

In the hybrid algorithms, we assume that the matrix to be inverted is in the GPU memory. This implies that there exists a preliminary transfer of the data from the (host) main memory to the device (GPU) memory. When presenting the update performed during the iteration, we will specify in which architecture is carried out each computation. Therefore, depending on where initially each operand is located, the exact data transfers between the host and device memory spaces can be easily derived.

2 Matrix sign function solvers with application in control

In state-space, a dynamic linear time-invariant (LTI) system is represented as:

$$\begin{array}{rcll} \dot{x}(t) & = & Fx(t) + Bu(t), & t > 0, \qquad x(0) = x^0, \\ y(t) & = & Cx(t) + Du(t), & t \geq 0, \end{array} \tag{2}$$

where $x^0 \in \mathbb{R}^n$ stands for the initial state of the system; and $x(t) \in \mathbb{R}^n, u(t) \in \mathbb{R}^m$ and $y(t) \in \mathbb{R}^m$ contain, respectively, the states, inputs and outputs of the system. Here $F \in \mathbb{R}^{n \times n}$, $B \in \mathbb{R}^{n \times m}$, $C \in \mathbb{R}^{p \times n}$, $D \in \mathbb{R}^{p \times m}$ and, usually, $m, p \ll n$.

Given a system of order n as in (2), *model order reduction* aims at finding an alternative LTI system of order $r \ll n$ which can accurately replace the original system in subsequent operations [1]. Balanced truncation (BT) is an efficient numerical method

for model reduction of large-scale LTI systems; see, e.g., [6]. The major computational step in BT requires the solutions $Y, Z \in \mathbb{R}^{n\times n}$ of the coupled Lyapunov equations $FY + YF^T = BB^T, \quad F^TZ + ZF = C^TC$, which can be obtained from sign (F) and a few minor other computations [18].

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 positive semidefinite, the solution of the *LQOC problem* associated with (2) is given by the stabilizing feedback control law $u(t) = -R^{-1}B^TXx(t) = -Kx(t),\ t \geq 0$, with $X \in \mathbb{R}^{n\times n}$ positive semidefinite. Under certain conditions [15], this feedback is given by the solution of the ARE $F^TX + XF - XBR^{-1}B^TX + C^TQC = 0$. In [18], it is shown that the solution X can be easily obtained from sign (H), with

$$H = \begin{pmatrix} F & BR^{-1}B^T \\ C^TQC & -F^T \end{pmatrix}. \tag{3}$$

The two problems defined above illustrate the need to compute the sign function of a matrix (F or H). When the number of states of the LTI system (2) is large, and the state matrix F is dense, it is therefore necessary to compute the inverse of a sequence of large-scale dense matrices. Note also that, when F is symmetric negative definite all the matrices inverted during the computation of sign (F) inherit this property. On the other hand, a simple reorganization of the blocks in (3) yields a symmetric indefinite matrix, which can be leveraged to invert only symmetric indefinite matrices during the Newton iteration for the matrix sign function [12].

3 Inversion of general matrices

In this section we review the inversion of general matrices via the LU factorization and GJE [13]. In both cases, the contents of A are overwritten with its inverse (i.e., they are in-place procedures). Besides, the two approaches require $2s^3$ flops.

3.1 LU factorization

Matrix inversion of a general matrix $A \in \mathbb{R}^{s\times s}$ via the LU factorization can be accomplished in three steps. First the matrix is decomposed as $A = LU$, where $L \in \mathbb{R}^{s\times s}$ and $U \in \mathbb{R}^{s\times s}$ are lower unit and upper triangular factors, respectively; the upper triangular factor is then explicitly inverted: $U \to U^{-1} = \bar{U}$; and finally, the (lower unit) triangular system $A^{-1}L = \bar{U}$ is solved for the sought-after inverse A^{-1}.

Figure 2 illustrates the updates performed at each one of the three steps for the inversion via the LU factorization. The complete algorithms are obtained by including these operations into the update box of the appropriate algorithmic skeleton in Figure 1. For each operation, we indicate in which architecture it is performed (host/CPU or device/GPU) and the type of numerical kernels involved (using the LAPACK/BLAS interface). Consider the LU factorization first (left). This algorithm proceeds TL→BR and the resulting factors L and U overwrite, respectively, the strictly lower and the upper triangular parts of A. The invocation to the kernel LU, in the first operation

TL→BR2 × 2. LU factorization		TL→BR2 × 2. $U \rightarrow U^{-1} = \bar{U}$		BR→TL2 × 2. Solve $A^{-1}L = \bar{U}$	
$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} := \text{LU} \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$	CPU/ GETRF	$A_{01} := \text{TRIU}(A_{00})A_{01}$	GPU/ TRMM	$\begin{pmatrix} W_1 \\ W_2 \end{pmatrix} := \text{TRILS} \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix}$	GPU/ LACPY
$A_{12} := \text{TRILU}(A_{11})^{-1}A_{12}$	GPU/ TRSM	$A_{01} := -A_{01}$ $\text{TRIU}(A_{11})^{-1}$	GPU/ TRSM	$\text{TRILS} \begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} := 0$	GPU/ LASET
$A_{22} := A_{22} - A_{21}A_{12}$	GPU/ GEMM	$\text{TRIU}(A_{11}) :=$ $\text{TRIU}(A_{11})^{-1}$	CPU/ TRTRI	$\begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \end{pmatrix} - := \begin{pmatrix} A_{02} \\ A_{12} \\ A_{22} \end{pmatrix} W_2$	GPU/ GEMM
				$\begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \end{pmatrix} := \begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \end{pmatrix} W_1^{-1}$	GPU/ TRSM

Figure 2: Operations (updates) to invert a general matrix A via the LU factorization. Left: LU factorization. Center: Inversion of the triangular factor U. Right: Solution of lower triangular system.

of the update, obtains the factorization of the *panel* $\left(A_{11}^T, A_{12}^T\right)^T$, of (column) width b. In our hybrid algorithm, this operation is performed in the CPU (via kernel GETRF). Since we assume that the matrix is initially stored in the GPU, this implies that the panel has to be transferred to the host memory prior to this computation. The other two operations involved in the factorization, a triangular system solve (TRSM) and a matrix-matrix product (GEMM), are performed in the GPU. Thus, the triangular matrices resulting from the panel factorization have to be transferred back to the GPU before these two operations commence.

Upon termination of the algorithm for the inversion of the triangular matrix U, in the center of the figure, the upper triangular part of A (which initially contained U) is overwritten with $\bar{U} = U^{-1}$. At each iteration, the algorithm computes the product of a triangular matrix times a general one (TRMM), the solution of an upper triangular linear system, and the inversion of the triangular $b \times b$ block A_{11} (TRTRI). The former two operations are performed in the GPU, while the last one is carried out by the CPU. Thus, A_{11} is transferred from the device to the host before its inversion, and the result is moved back to the GPU. Finally, the solution of the triangular system $A^{-1}L = \bar{U}$ for A^{-1} (right) performs a matrix-matrix product and a triangular system solve per iteration, both computed in the GPU. Thus, the algorithm requires no data transfers. For high performance, though, this operation employs a small workspace, of size $s \times b$, in general with $b \geq 128$.

The previous algorithms illustrate a pattern common to most blocked procedures for dense linear algebra operations. The matrix is processed by blocks of a certain dimension (block size) and, at each iteration, only a part of the matrix is updated. In the hybrid CPU-GPU algorithms, we employ the GPU for the computationally-intensive, data-parallel operations, leaving the CPU for the kernels with strong control dependencies. This in turn dictates the data movements during the algorithm's iterations. For high performance, our hybrid algorithms will pursue two goals: 1) amortize the

TL→BR2 × 2. Invert general A via GJE		TL→BR2 × 2. Invert s.p.d. A via GJE	
$\begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \end{pmatrix} := \text{GJE} \begin{pmatrix} A_{01} \\ A_{11} \\ A_{21} \end{pmatrix}$	CPU/–	$\text{TRIU}(A_{11}) := \text{CHOL}(A_{11})$	CPU/POTRF
$A_{00} := A_{00} + A_{01}A_{10}$	GPU/GEMM	$\text{TRIU}(A_{11}) := \text{TRIU}(A_{11})^{-1}$	CPU/TRTRI
$A_{20} := A_{20} + A_{21}A_{10}$	GPU/GEMM	$A_{01} := A_{01}\text{TRIU}(A_{11})^T$	GPU/TRMM
$A_{10} := A_{11}A_{10}$	GPU/GEMM	$\text{TRIU}(A_{00}) := \text{TRIU}(A_{00} + A_{01}A_{01}^T)$	GPU/SYRK
$A_{02} := A_{02} + A_{01}A_{12}$	GPU/GEMM	$A_{01} := A_{01}\text{TRIU}(A_{11})$	GPU/TRMM
$A_{22} := A_{22} + A_{21}A_{12}$	GPU/GEMM	$A_{12} := \text{TRIU}(A_{11})^{-T}A_{12}$	GPU/TRSM
$A_{12} := A_{11}A_{12}$	GPU/GEMM	$\text{TRIU}(A_{22}) := \text{TRIU}(A_{22} - A_{12}^T A_{12})$	GPU/SYRK
		$A_{02} := A_{02} - A_{01}A_{12}$	GPU/GEMM
		$A_{12} := -\text{TRIU}(A_{11})A_{12}$	GPU/TRMM
		$\text{TRIU}(A_{11}) := \text{TRIU}(A_{11})\text{TRIU}(A_{11})^T$	CPU/LAUUM

Figure 3: Operations (updates) to invert a general or a s.p.d. matrix A via GJE (left or right, respectively.)

amount of data that are transferred between CPU and GPU with a large number of flops (per iteration); and 2) cast most of the flops in terms of a reduced number of high performance GPU kernels operating on large pieces of data. Consider, e.g., the algorithm for the LU factorization. At each iteration, a panel of $k \times b$ elements is transferred between CPU and GPU (k initially equals s and decreases with each iteration by a factor of b), while this overhead has to be amortized roughly by the $2(k-b)^2 b$ flops corresponding to the update of A_{22}. Also, the major part of the computation is cast in terms of the matrix-matrix product for the update of A_{22}, an operation that is well-known to attain high performance on most current architectures and, in particular, on GPUs.

3.2 Gauss-Jordan elimination

An alternative blocked procedure to invert a matrix via GJE is shown in Figure 3 (left) [17]. Compared with the three sweeps of the inversion approach based on the LU factorization (one per step), the algorithm computes the inverse in a single sweep of the matrix and is richer in large matrix-matrix products. All operations are performed on the GPU, except for the factorization of the current panel $\left(A_{01}^T, A_{11}^T, A_{12}^T, \right)^T$ (kernel GJE), which is done in the CPU. Thus, at the beginning of each iteration, this panel of dimension $s \times b$ is transferred to the CPU, processed there, and the results are sent back to the GPU. Note the regularity of the algorithm: All iterations perform the same number of flops (approximately, $2sb(s-b)$ flops) and the same amount of data is transferred between host and device (except for the last iteration if the problem size is not an exact multiple of b).

4 Inversion of s.p.d. matrices

We next describe the specialized counterparts of the LUpp- and GJE-based methods for the inversion of s.p.d. matrices. These alternatives exploit the symmetric structure of the matrix $A \in \mathbb{R}^{s \times s}$ to halve its computational cost: from the $2s^3$ flops of the general case to roughly s^3 flops in this case. The variants presented next compute the inverse in-place, requiring no additional workspace. In fact, in the algorithms, the upper triangular part of the matrix is replaced by the upper triangle of its inverse while the strictly lower part is not referenced/modified. Also, the symmetry combined with the positive definiteness render unnecessary the application of pivoting for numerical stability in the algorithms.

4.1 Cholesky factorization

This algorithm performs the inversion in three steps (sweeps). First, the s.p.d. matrix is decomposed into the product $A = U^T U$, where the upper triangular matrix $U \in \mathbb{R}^{s \times s}$ is the Cholesky factor. Next, the Cholesky factor is explicitly inverted $U \rightarrow U^{-1} = \bar{U}$ and, finally, the inverse is obtained as $A^{-1} := \bar{U}\bar{U}^T = U^{-1}U^{-T}$.

Figure 4 reports the updates performed in the first and third steps (Cholesky factorization and matrix product $A^{-1} := U^{-1}U^{-T}$, respectively) of matrix inversion via the Cholesky factorization. The inversion of the triangular factor U (second step) can be obtained using the same algorithm described in the previous section. At each iteration of the Cholesky factorization (left column of the figure), the $b \times b$ block A_{11} is first transferred to the CPU, factorized there (via a call to kernel POTRF), and the triangular factor is moved back to the GPU. This factor is then used to solve a triangular system and perform a symmetric rank-b update (SYRK) of the trailing submatrix A_{22} in the GPU. It is important to realize that only the upper triangular part of A_{22} is updated, which in practice halves the cost of this factorization with respect to that of the general case. This is attained by using the appropriate kernel for the update of A_{22}, which requires only $(k-b)^2 b$ flops (with $k := s$ initially, and decreasing by a factor of b per iteration).

At the beginning of the third step (Figure 4 (right)), $\bar{U} = U^{-1}$ overwrites the upper triangular part of A and, therefore, the matrix product that is computed corresponds to $A^{-1} := \bar{U}\bar{U}^T = U^{-1}U^{-T}$. The algorithm consists of four operations: the product of a general matrix times a triangular one, a call to compute the (in-place) inverse of the $b \times b$ block A_{11} (LAUUM), a general matrix-matrix product, and a symmetric rank-b update. All these operations, except the second one, are performed in the GPU. This dictates the need to transfer the corresponding $b \times b$ block between the GPU and the CPU before the second operation, and in the opposite direction immediately after it.

4.2 Gauss-Jordan elimination

The procedure for matrix inversion of s.p.d. matrices based on GJE is illustrated in Figure 3 (right) [9]. The algorithm consists of a sequence of 10 operations of different types: Cholesky factorization, triangular matrix inversion, matrix multiplications with

TL→BR2 × 2. Cholesky factorization		TL→BR2 × 2. $A^{-1} := \bar{U}\bar{U}^T$	
$\text{TRIU}(A_{11}) := \text{CHOL}(A_{11})$	CPU/POTRF	$A_{01} := A_{01}\text{TRIU}(A_{11})^T$	GPU/TRMM
$A_{12} := \text{TRIU}(A_{11})^{-1}A_{12}$	GPU/TRSM	$\text{TRIU}(A_{11}) := \text{TRIU}(A_{11})\text{TRIU}(A_{11})^T$	CPU/LAUUM
$\text{TRIU}(A_{22}) := \text{TRIU}(A_{22} - A_{12}^T A_{12})$	GPU/SYRK	$A_{01} := A_{01} + A_{02}A_{12}^T$	GPU/GEMM
		$\text{TRIU}(A_{11}) := \text{TRIU}(A_{11} + A_{12}A_{12}^T)$	GPU/SYRK

Figure 4: Operations (updates) to invert a s.p.d. matrix A via the Cholesky factorization. Left: Cholesky factorization. Right: Triangular matrix-matrix multiplication $A^{-1} := \bar{U}\bar{U}^T$.

TL→BR2 × 2. LDL^T factorization		BR→TL2 × 2. Invert symmetric indefinite A	
$\left[\begin{pmatrix} A_{11} \\ A_{12} \end{pmatrix}, W\right] := LDLt\begin{pmatrix} A_{11} \\ A_{12} \end{pmatrix}$	CPU/–	$W_2 := \bar{D}_2 A_{21}$	CPU/–
$\text{TRIL}(A_{22}) := \text{TRIL}(A_{22} - A_{21}W)$	CPU/GEMV,GEMM	$\text{TRIL}(W_1) := \text{TRIL}(\bar{D}_1 A_{11})$	CPU/–
		$W_1 := \text{TRIL}(A_{11})^T W_1$	GPU/TRMM
		$\text{TRIL}(A_{11}) := \text{TRIL}(W_1)$	GPU/LACPY
		$W_1 := A_{21}^T W_2$	GPU/GEMM
		$\text{TRIL}(A_{11}) := \text{TRIL}(A_{11} + W_1)$	GPU/–
		$A_{21} := \text{TRIL}(A_{22})^T W_2$	GPU/TRMM

Figure 5: Operations (updates) to invert a symmetric indefinite matrix A via the LDL^T factorization. Left: LDL^T factorization. Right: Inversion $A^{-1} := L^{-T}D^{-1}L^{-1}$.

operands of diverse structures, triangular system solve, and symmetric rank-b update. All operations except the first two and the last one are performed in the GPU. Therefore, the algorithm requires a couple of transfers of the $b \times b$ block A_{11} between GPU and CPU and back per iteration.

5 Inversion of symmetric indefinite matrices

Symmetric indefinite matrices can be inverted in-place, at a cost of s^3 flops, following a three-step (-sweep) procedure (though, for high performance, an additional $s \times b$ array is required.) First the matrix is decomposed as the product $A = LDL^T$, where $L \in \mathbb{R}^{s\times s}$ is lower triangular and $D \in \mathbb{R}^{s\times s}$ is diagonal with 1×1 or 2×2 diagonal blocks [13]; then, the lower triangular and diagonal factors are explicitly inverted: $L \to L^{-1} = \bar{L}$ and $D \to D^{-1} = \bar{D}$; and finally, the inverse is obtained from $A^{-1} := \bar{L}^T\bar{D}\bar{L} = L^{-T}D^{-1}L^{-1}$. In practice, the LDL^T factorization includes pivoting to attain numerical stability close to that of partial pivoting (not included in the following presentation, for simplicity). Also, only the contents of the lower triangular part of A are overwritten with the lower triangle of the inverse.

Figure 5 introduces a simplified version of the updates performed in the algorithms

for the first and third steps. The second step is performed following a transposed variant of the algorithm to invert an upper triangular matrix (see Section 3). The procedure in the left of the figure computes the factorization. To do this, at each iteration it obtains the entries of the lower triangular factor L corresponding to the panel $\left(A_{11}^T, A_{12}^T\right)^T$ while, simultaneously, building the product of this factor times the corresponding diagonal blocks into W. Because of the symmetric structure of the matrix, and the need to reference only its lower triangle, the computation of this panel factorization is quite elaborate. The update of the trailing submatrix A_{22} also needs to be carefully done. In particular, to reduce the arithmetic cost, to avoid modifying the contents in the strictly upper triangular part of the array A_{22}, and to still deliver fair performance, the update proceeds by panels of b columns. At each iteration, the upper $b \times b$ block of this panel is updated column-wise, using (repeated invocations to) kernel GEMV (one per column), while the subdiagonal block of the panel is performed with a single invocation to kernel GEMM and comprises the major part of the computational cost of the update. The algorithm in the right of the figure computes the inverse of A from its LDL^T factorization. Assume that the lower triangular part of A already contains $\bar{L} = L^{-1}$ and consider a partitioning of $\bar{D} = D^{-1} = \text{DIAG}(\bar{D}_0, \bar{D}_1, \bar{D}_2)$ conformal with that of A. Then, at each iteration, the algorithm performs two triangular matrix-matrix multiplications (to update the workspace W_1 and A_{21}) and a matrix-matrix product (to update A_{11}) on the GPU plus a few additional minor operations.

6 Optimizing the hybrid execution of matrix inversion kernels

In this section we describe several techniques which, in general, render higher performance if incorporated into the basic hybrid CPU-GPU algorithms for matrix inversion presented in the previous three sections. For clarity, when necessary we will illustrate the corresponding technique using the GJE approach for matrix inversion of general matrices in Figure 3 (left).

Padding. Previous studies [2] have demonstrated the performance benefits of utilizing GPU kernels that operate with arrays of dimension(s) that are integer multiples of 32. This is due to the number of arithmetic units in current GPUs and, when employing dense linear algebra libraries of GPU kernels, to the routines in these libraries being specially optimized for such problem sizes. For other problem dimensions, one simple solution is to "pad" the arrays to the next integer multiple of 32. For the inversion of $A \in \mathbb{R}^{s \times s}$, this can be easily done by embedding the data of the matrix into an array of dimension $v \times v = (s + r) \times (s + r)$, with $0 \leq r < 32$ and v an exact multiple of 32, initialized to the identity matrix. When s is moderately large, the overhead incurred by padding is negligible while the benefits clearly pay off.

Blocked kernels for the CPU. Most of the algorithms presented earlier involved a sort of "recursive" call to process a block or a panel of the matrix in the CPU. In the case of the GJE inversion algorithm, e.g., this is the call to GJE to factorize the

current $s \times b$ panel. To enhance performance of GPU kernels, in practice b is often chosen to be large, around 512 or even larger. Therefore, it is usually recommended to perform the execution of the CPU operation using a blocked algorithm, with a smaller block size, let's say $\bar{b} \approx 128 < b$, and a highly tuned, possibly multi-threaded CPU implementation for the operation, e.g., from LAPACK or BLAS.

Overlapping communication and computation (double-buffering). In the presentation of the inversion algorithms, we assumed that the matrix was already stored on the GPU, which required an initial transfer of the matrix contents from the main memory to the device. This overhead can certainly be hidden by employing *double-buffering* so that the transfer of the data proceeds concurrently with the arithmetic operations. However, given that our matrix inversion algorithms are repeatedly invoked from the Newton iteration (1), we can neglect the costs of the initial transfer of the data between the main memory (in the host) and the GPU memory and the final retrieval of the result from device to the host. While double-buffering can also be employed to hide the latency of the data transfers that occur inside the iteration loop, in most cases the cost of this transfer is negligible compared with the actual number of operations performed during the iteration.

Look-ahead. The algorithms for the LU factorization, the GJE for general matrices, and the LDL^T factorization operate on a large-and-narrow panel, which stands in the critical path of the computation and limits the concurrency of the algorithm. For example, in the GJE the current panel has to be factorized before any of the matrix products can proceed. The next panel then presents a critical point, which marks a new synchronization point. As the number of computational resources grows, the factorization (which is basically sequential or, at least, offers a much more limited degree of parallelism than the rest of the operations) becomes a bottleneck. One partial solution to this is to overlap the factorization of the $k+1$-th panel with the updates associated with the factorization of the current one. Specifically, consider the factorization of the k-th panel is done resulting in b "Gauss-Jordan transforms". We can then update only the columns in the $k+1$-st panel, and perform the factorization of this panel concurrently with the update of the remaining part of the matrix with the previous Gauss-Jordan transforms. This technique is known as "look-ahead" and can be applied with different degrees of depth [19].

Concurrent CPU-GPU execution. Most hybrid platforms are equipped with a powerful GPU but also an appreciable computational power associated with one or more general-purpose multi-core processors. Clearly more performance can be attained if both resources are employed for the computation of the algorithms involved in matrix inversion, and specially, if the two resources can operate in parallel. Given that GPU kernels are asynchronous, this can be e.g. combined with the look-ahead technique just described.

7 Experimental Results

In the following, we perform an experimental evaluation of several *ad-hoc* routines for matrix inversion based on the hybrid CPU-GPU algorithms introduced in Sections 3–5,

optimized with the techniques described in Section 6. For comparison, our study also includes numerical kernels from diverse dense linear algebra libraries for multi-core processors, many-core GPUs, and hybrid architectures:

- Intel's MKL (v11.1) implements LAPACK and BLAS for multi-core architectures (results were obtained using 8 threads). For those LAPACK-level routines that are not in MKL, we employed the legacy codes available at `www.netlib.org/lapack` (v3.4).
- NVIDIA's CUBLAS (v4.0) provides BLAS for single-GPU systems.
- EM Photonics' CULA (R13a) is a commercial product which partially implements LAPACK and BLAS for hybrid platforms equipped with a single GPU.
- UTK's MAGMA (v1.1.0), like the previous case, implements LAPACK and BLAS partially, targeting the same class of hybrid platforms.

Hereafter, we will refer to the ad-hoc implementations as the BASIC variants. Routines from CUBLAS, CULA or MAGMA can be expected to include advanced optimization techniques to deliver similar or higher performance than the basic ones. The experiments do not evaluate UT's FLAME, because the runtime in the `libflame` library is tailored for multi-GPU platforms.

All experiments were performed on a platform consisting of two INTEL XEON E5520 quad-core processors (8 cores) at 2.27GHz, connected to an NVIDIA Tesla C2050, using IEEE double-precision arithmetic. All operands were always square, with their dimensions being integer multiples of 32 as padding ensures similar efficiency for other problem sizes. Performance is compared in terms of GFLOPS (10^9 flops/secs.), using the standard flops count for each operation. The cost of data communication between the CPU and the GPU memories is always counted, except for the initial transfer of the original matrix to the GPU and the final retrieval of the result (see Section 6).

Let us start with the inversion of general matrices (see Section 3). Our first experiment, with results in Figure 6, evaluates the performance attained by the operations corresponding to the three steps for the inversion of general matrices based on the LU factorization with partial pivoting (LUpp), and the complete matrix inversion procedures via LUpp and GJE. Specifically, the plot in the top-left corner compares the performance attained by the implementations for LUpp in MKL, MAGMA, CULA; and a BASIC variant which employs kernels from CUBLAS and MKL. The routine from MAGMA is the fastest one for larger matrices, while the CULA variant attains higher performance for matrices of dimension up to 2,000. Similarly, the plot in the top-right corner displays the results obtained by four implementations for the triangular matrix inversion, with the BASIC variant clearly outperforming the routines in MKL, MAGMA and CULA. The plot in the bottom-left corner illustrates the performance of the kernels for the solution of lower triangular systems provided by CUBLAS, CULA, MAGMA and MKL, reporting MAGMA as the best choice. Finally, the plot in the bottom-right corner compares the efficiency obtained by the fastest implementations of the complete LUpp-based and GJE-based matrix inversion algorithms including,

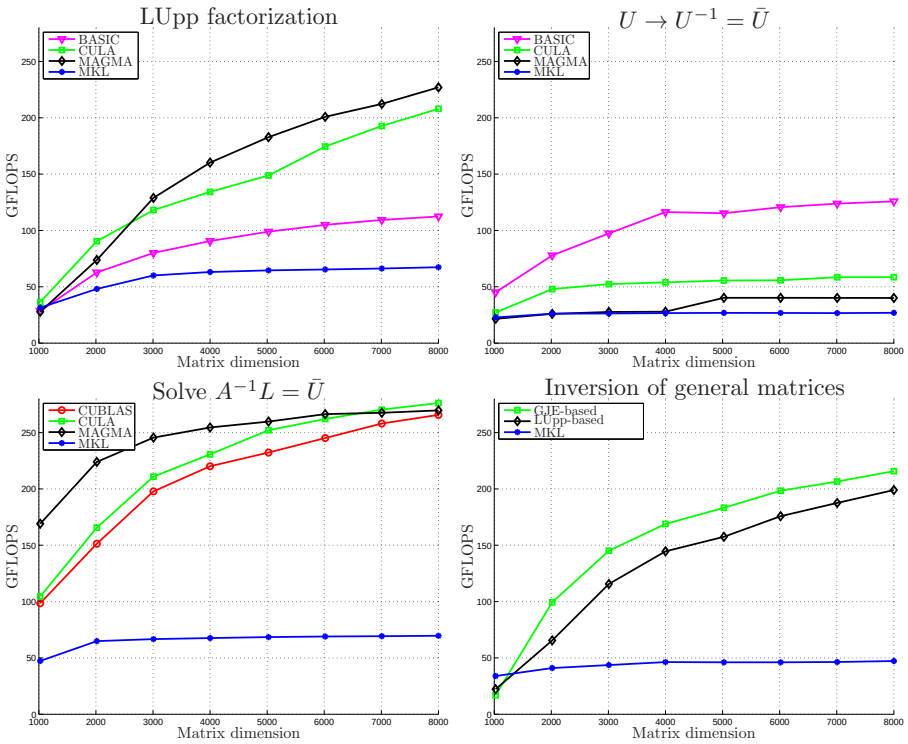

Figure 6: Inversion of general matrices. Top-left: LUpp; top-right: triangular matrix inversion; bottom-left: solution of lower triangular systems; bottom-right: Complete algorithm.

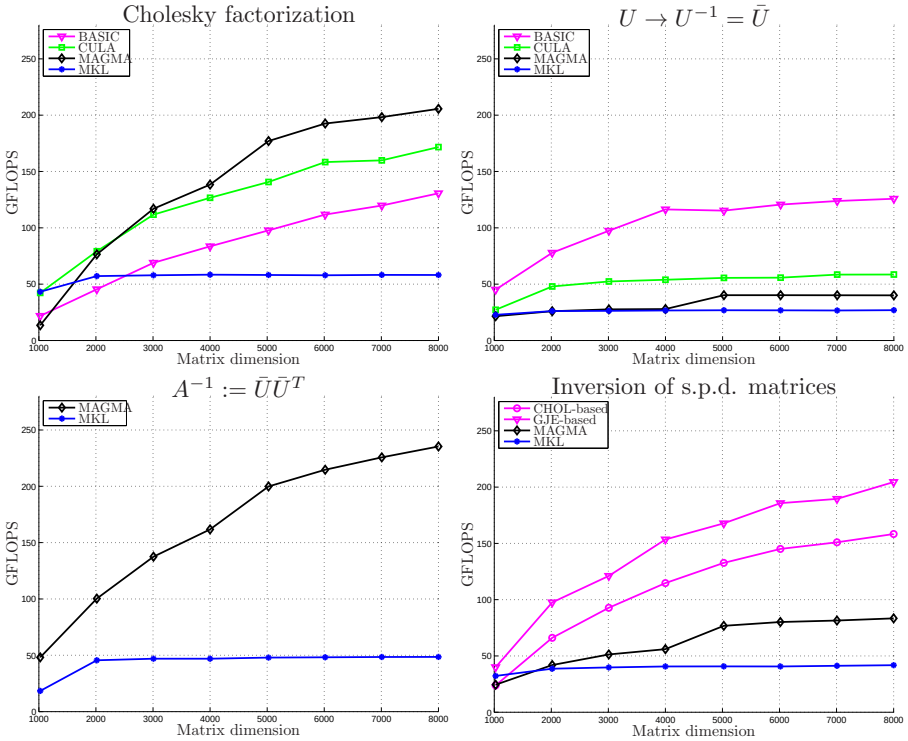

Figure 7: Inversion of s.p.d. matrices. Top-left: Cholesky factorization; top-right: triangular matrix inversion; bottom-left: multiply $A^{-1} := \bar{U}\bar{U}^T$; bottom-right: Complete algorithm.

for reference, MKL (routines DGETRF+DGETRI). In this case, the GJE-based implementation is a hybrid variant that employs the GEMM kernel from MAGMA for the matrix-matrix products on the GPU while all other computations are performed on the CPU using kernels from MKL. The LUpp-based routine is also an ad-hoc implementation which employs MAGMA for the LUpp and the solution of the lower triangular system, and the BASIC variant for the inversion of the triangular factor U. The last plot demonstrates the superior performance of the GJE-based inversion algorithm on the hybrid CPU-GPU architecture.

Consider next the inversion of s.p.d. matrices (see Section 4). Figure 7 summarizes the results obtained for the two inversion algorithms of this class of matrices and the operations involved in the procedure based on the Cholesky factorization. The results in the top-left plot correspond to the computation of the Cholesky factorization, using routines from CULA, MAGMA and MKL, as well as an ad-hoc BASIC variant that employs kernels from CUBLAS. The best results are obtained by the MAGMA routine for matrices larger than 2,000, while for smaller problems, the CULA and MKL imple-

mentations are faster. The second step of the algorithm, as in the general matrix case, requires the inversion of an upper triangular matrix. The results in the top-right plot thus replicate those in the figure for the general case. The plot in the bottom-left corner rehearses the performance of the implementation in MAGMA for the computation $A^{-1} := \bar{U}\bar{U}^T$, including MKL for reference. (No other implementation is available, at the moment, for this particular operation). The plot in the bottom-right corner shows the GFLOPS obtained with the Cholesky-based and the GJE-based approaches. In particular, the first variant utilizes the MAGMA kernels for the computation of the Cholesky factorization and the product $\bar{U}\bar{U}^T$, and the BASIC routine for the inversion of the triangular factor. The alternative procedure, based on the GJE method, employs kernels from CUBLAS for the major GPU operations, and MKL for other minor operations, like the factorization or the inversion of small blocks. As in the general case, the best performance is obtained with the GJE algorithm.

Let us finally move to the third case, the inversion of symmetric indefinite matrices (see Section 5). Among the libraries considered, only MKL implements the full procedure for the inversion of this class of matrices. The algorithm for the first stage (computation of the factorization $A = L^T DL$) is hardly suitable for many-core architectures, due to the intricacies of the application of pivoting in the symmetric case. This also explains the low performance obtained by MKL in the multi-core architecture: less than 30 GFLOPS for the larger matrices; see left plot in Figure 8. For this particular operation, we have developed an optimized version for multi-threaded CPUs, which merges the updates performed to each panel of A_{22} into a single call to GEMM, and extracts parallelism by distributing the iteration space among the cores using a simple OpenMP `pragma omp parallel for`, yielding a higher GFLOPS rate. The second stage is primarily dedicated to the inversion of the (lower) triangular factor, and can be accomplished using a transposed variant of our basic routine for the inversion of an upper triangular factor. (The inversion of D, also carried out in this stage, is straight-forward.) The third stage ($A^{-1} := \bar{L}^T \bar{D}\bar{L} = L^{-T}D^{-1}L^{-1}$) exhibits higher concurrency and can be executed in a GPU with moderate efficiency. The left-hand side plot in the figure also collects the performance obtained for the second+third stages (lines labelled as $A^{-1} := L^{-T}D^{-1}L^{-1}$) by the routine from MKL and a BASIC implementation that employs kernels from MKL and CUBLAS. The hybrid variant clearly outperforms MKL for larger matrices, but it hardly attains 70 GFLOPS. The right-hand side plot shows the results for the inversion of symmetric indefinite matrices. There, the BASIC variant employs our tuned CPU factorization (DSYTRF) and the ad-hoc hybrid implementation to obtain the inverse from the factors. This implementation outperforms the MKL+LAPACK code (DSYTRF from MKL and DSYTRI2X from LAPACK) for matrices of dimension larger than 2,000. However, its performance is poor if we compare it, e.g., with that observed for the inversion of general matrices (see Figure 6). Consequently, with the implementations available at the moment, it is faster to calculate the inverse of a symmetric indefinite matrix using the general matrix inversion algorithm, even if that implies doubling the number of flops.

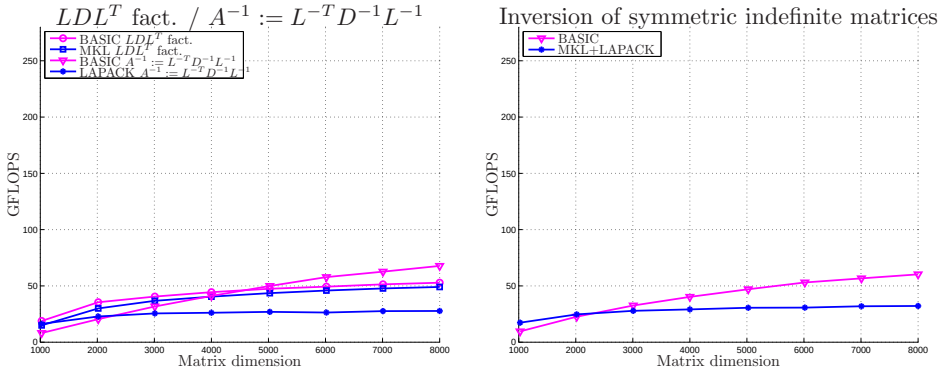

Figure 8: Inversion of symmetric indefinite matrices (right) and the computation of the inverse of a matrix from its LDL^T factorization (left).

8 Concluding Remarks

In this paper we have studied the inversion of large-scale, dense matrices, on hybrid CPU-GPU platforms, with application to the solution of matrix equations arising in control theory. Borrowing the notation of the FLAME project, we have presented several matrix inversion algorithms, based on common matrix factorizations and the GJE method, for general, s.p.d. and symmetric indefinite problems. The performance of our practical implementations of these algorithms has been compared with analogous kernels from CPU and hybrid CPU-GPU dense linear algebra libraries, on a representative high performance platform.

The results for the general and s.p.d. case on this architecture expose the performance advantage of the GJE-based matrix inversion methods over their LU- and Cholesky-based counterparts. Combined with their extreme simplicity, the result is a wide-appeal GJE-based approach to leverage the hardware concurrency of current hybrid platforms. As for the symmetric indefinite case, the need to preserve the structure of the problem during the application of pivoting stands in the way of an efficient concurrent implementation of the method so that further work remains to be done to turn this into a competitive method for tackling this class of problems.

References

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

[2] S. Barrachina, M. Castillo, F.D. Igual, R. Mayo, E.S. Quintana-Ortí, and G. Quintana-Ortí. Exploiting the capabilities of modern GPUs for dense matrix computations. *Conc. Comp.: Prac. Exp.*, 21:2457–2477, 2009.

[3] P. Benner, P. Ezzatti, D. Kressner, E.S. Quintana-Ortí, and A. Remón. A mixed-precision algorithm for the solution of lyapunov equations on hybrid CPU-GPU platforms. *Parallel Comp.*, 37(8):439–450, 2011.

[4] P. Benner, P. Ezzatti, D. Kressner, E.S. Quintana-Ortí, and A. Remón. Accelerating model reduction of large linear systems with graphics processors. In *Lecture Notes in Computer Science, State of the Art in Scientific and Parallel Computing*, volume 7134, pages 88–97. Springer, 2012. (on-line version available).

[5] P. Benner, P. Ezzatti, E.S. Quintana-Ortí, and A. Remón. High performance matrix inversion of SPD matrices on graphics processors. In *Workshop on Exploitation of Hardware Accelerators–WEHA 2011*, pages 640–646. IEEE, 2011.

[6] P. Benner, E.S. Quintana-Ortí, and G. Quintana-Ortí. State-space truncation methods for parallel model reduction of large-scale systems. *Parallel Computing*, 29:1701–1722, 2003.

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

[8] P. Bientinesi, J.A. Gunnels, M.E. Myers, E.S. Quintana-Ortí, and R.A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Trans. Math. Soft.*, 31(1):1–26, March 2005.

[9] P. Bientinesi, B. Gunter, and R. A. van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Softw.*, 35:3:1–3:22, July 2008.

[10] CULA project home page. `http://www.culatools.com/`.

[11] P. Ezzatti, E. Quintana-Ortí, and A. Remón. Using graphics processors to accelerate the computation of the matrix inverse. *The Journal of Supercomputing*, pages 1–9, 2011. 10.1007/s11227-011-0606-4.

[12] J.D. Gardiner and A.J. Laub. A generalization of the matrix-sign-function solution for algebraic Riccati equations. *Internat. J. Control*, 44:823–832, 1986.

[13] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[14] J.A. Gunnels, F.G. Gustavson, G.M. Henry, and R.A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[15] P. Lancaster and L. Rodman. *The Algebraic Riccati Equation*. Oxford University Press, Oxford, 1995.

[16] MAGMA project home page. `http://icl.cs.utk.edu/magma/`.

[17] E.S. Quintana-Ortí, G. Quintana-Ortí, X. Sun, and R.A. van de Geijn. A note on parallel matrix inversion. *SIAM J. Sci. Comput.*, 22:1762–1771, 2001.

[18] J.D. Roberts. 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).

[19] P. Strazdins. 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.

[20] University of Texas, `http://www.cs.utexas.edu/~flame/`.