# Accelerating Band Linear Algebra Operations on GPUs with Application in Model Reduction

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

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

**Abstract.** In this paper we present new hybrid CPU-GPU routines to accelerate the solution of linear systems, with band coefficient matrix, by off-loading the major part of the computations to the GPU and leveraging highly tuned implementations of the BLAS for the graphics processor. Our experiments with an nVidia S2070 GPU report speed-ups up to 6× for the hybrid band solver based on the LU factorization over analogous CPU-only routines in Intel's MKL. As a practical demonstration of these benefits, we plug the new CPU-GPU codes into a sparse matrix Lyapunov equation solver, showing a 3× acceleration on the solution of a large-scale benchmark arising in model reduction.

**Key words:** Band linear systems, linear algebra, graphics processors (GPUs), high performance, control theory.

## 1 Introduction

Linear systems with band coefficient matrix appear in a large variety of applications, including finite element analysis in structural mechanics, domain decomposition methods for partial differential equations in civil engineering, and as part of matrix equations solvers in control and systems theory. Exploiting the structure of the matrix in these problems yields huge savings, both in number of computations and storage space. This is recognized by LAPACK [1, 2] that, when linked to a (multi-threaded) implementation of BLAS, provides an efficient means to solve band linear systems on general-purpose (multicore) processors.

In the last few years, hybrid computer platforms consisting of multicore processors and GPUs (graphics processing units) have evolved from being present only in a reduced niche, to become common in many application areas with high computational requirements [3]. A variety of reasons have contributed to the progressive adoption of GPUs, including the introduction of nVidia's programming framework CUDA [4, 5] and the OpenACC application programming

interface (API), combined with impressive raw performance, affordable price, and an appealing power-performance ratio. In particular, in the area of dense linear algebra many studies have now demonstrated remarkable performance improvements by using GPUs; see e.g., among many others, [6–8].

In this paper we present new LAPACK-style routines that leverage the large-scale hardware parallelism of hybrid CPU-GPU platforms to accelerate the solution of band linear systems. In particular, the experimental results collected on a hardware platform equipped with an Intel i7-2600 processor and an NVIDIA S2070 ("Fermi") GPU with the accelerator-enabled codes demonstrate superior performance and scalability over the highly-tuned multithreaded band solver in Intel MKL (Math Kernel Library). Furthermore, the integration of the GPU solvers with Lyapack [9], a library for the solution of linear and quadratic matrix equations, reveals that these benefits carry over to the solution of model reduction problems arising in control theory.

The rest of the paper is structured as follows. In Section 2 we review the LAPACK routines for the solution of band linear systems, while in Section 3 we introduce our new hybrid CPU-GPU routines for band matrix factorization and the solution of triangular band systems in detail. Section 4 summarizes the experimental evaluation of our new solvers and Section 5 analyzes the application of the developed kernels to the solution of linear matrix equations arising in model reduction (specifically, sparse Lyapunov equations). Finally, a few concluding remarks and a discussion of open questions close the paper in Section 6.

## 2 Solution of Band Linear Systems with LAPACK

The solution of a linear system of the form

$$A\,X = B, \tag{1}$$

where $A \in \mathbb{R}^{n\times n}$ is a band matrix with upper and lower bandwidth $k_u$ and $k_l$ respectively, $B \in \mathbb{R}^{n\times m}$ contains a collection of $m$ right-hand side vectors (usually with $m \ll n$), and $X \in \mathbb{R}^{n\times m}$ is the sought-after solution can be performed in two steps using LAPACK. First, the coefficient matrix $A$ is decomposed into two triangular band factors $L, U \in \mathbb{R}^{n\times n}$ (LU factorization) using the routine GBTRF. Then, $X$ is obtained by solving two triangular band systems with coefficients $L$ and $U$ using the routine GBTRS. In this section we describe the process implemented in LAPACK and point out some of the drawbacks of the corresponding routines.

### 2.1 Factorization of band matrices

LAPACK includes two routines for the computation of the LU factorization of a band matrix: GBTF2 and GBTRF, which encode, respectively, unblocked and blocked algorithmic variants of the operation. The former performs most of the computations in terms of BLAS-2 operations, while GBTRF is rich in BLAS-3

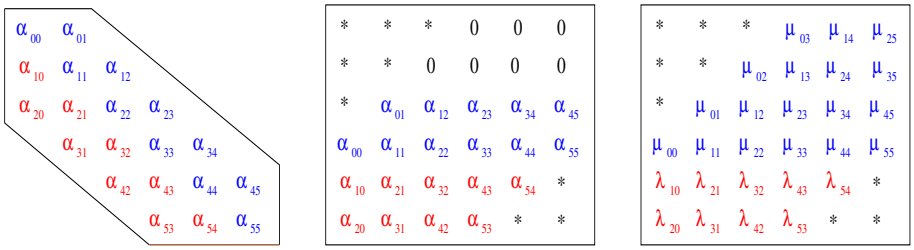


**Fig. 1.** $6 \times 6$ band matrix with upper and lower bandwidths $k_l = 2$ and $k_u = 1$, respectively (left); packed storage scheme used in LAPACK (center); result of the LU factorization where $\mu_{i,j}$ and $\lambda_{i,j}$ stand, respectively, for the entries of the upper triangular factor $U$ and the multipliers of the Gauss transforms.

operations. As a consequence, GBTRF is more efficient for large matrices, like those appearing in our control applications. Therefore we will focus hereafter on that particular algorithmic variant.

Routine GBTRF computes the *LINPACK-style LU factorization with partial pivoting*

$$L_{n-2}^{-1} \cdot P_{n-2} \cdots L_1^{-1} \cdot P_1 \cdot L_0^{-1} \cdot P_0 \cdot A = U \tag{2}$$

where $P_0,\ P_1, \ldots, P_{n-2} \in \mathbb{R}^{n \times n}$ are permutation matrices, $L_0,\ L_1, \ldots, L_{n-2} \in \mathbb{R}^{n \times n}$ represent Gauss transforms, and $U \in \mathbb{R}^{n \times n}$ is upper triangular with upper bandwidth $k_l + k_u$. Figure 1 illustrates the packed storage scheme used for band matrices in LAPACK and how this layout accommodates the result of the LU factorization with pivoting. We note there that $A$ is stored with $k_l$ additional superdiagonals initially set to zero to make space for fill-in due to pivoting during the factorization. Upon completion, the entries of the upper triangular factor $U$ overwrite the upper triangular entries of $A$ plus these $k_l$ additional rows in the packed array, while the strictly lower triangular entries of $A$ are overwritten with the multipliers which define the Gauss transforms.

Assume that the algorithmic block size, $b$, internally employed in the blocked routine GBTRF, is an integer multiple of both $k_l$ and $k_u$, and consider the partitioning

$$A = \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \tag{3}$$

where $A_{TL}, A_{00} \in \mathbb{R}^{k \times k}$, (with $k$ an integer multiple of $b$,) $A_{11}, A_{33} \in \mathbb{R}^{b \times b}$, and $A_{22} \in \mathbb{R}^{l \times u}$, with $l = k_l - b$ and $u = k_u + k_l - b$.

Routine GBTRF encodes a right-looking factorization procedure; that is, an algorithm where, before iteration $k/b$ commences, $A_{TL}$ has been already factorized; $A_{ML}$ and $A_{TM}$ have been overwritten, respectively, by the multipliers and

the corresponding block of $U$; $A_{MM}$ has been correspondingly updated; and the rest of the blocks remain untouched. We note that, with this partitioning, $A_{31}$ is upper triangular while $A_{33}$ is lower triangular. Furthermore, at this point, $A_{13}$, $A_{23}$, $A_{24}$, and $A_{34}$ contain only zeros.

In order to move the computation forward by $b$ rows/columns, during the current iteration of the routine the following operations are performed (the annotations to the right of some of these operations correspond to the name of the BLAS routine that is used):

1. Obtain $W_{31} := \text{TRIU}(A_{31})$, a copy of the upper triangular part of $A_{31}$; and compute the LAPACK-style LU factorization with partial pivoting
$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \\ W_{31} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \\ L_{31} \end{pmatrix} U_{11}. \tag{4}$$
The blocks of $L$ and $U$ overwrite the corresponding blocks of $A$ and $W_{31}$. (In the actual implementation, the copy $W_{31}$ is obtained while this factorization is being computed.)
2. Apply the permutations in $P_1$ to the remaining columns of the matrix:
$$\begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} := P_1 \begin{pmatrix} A_{12} \\ A_{22} \\ A_{32} \end{pmatrix} \quad \text{and} \quad (\text{LASWP}) \tag{5}$$
$$\begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix} := P_1 \begin{pmatrix} A_{13} \\ A_{23} \\ A_{33} \end{pmatrix}. \tag{6}$$
A careful application of permutations is needed in (6) as only the lower triangular part of $A_{13}$ is physically stored. As a result of the application of permutations, $A_{13}$, which initially equals zero, may become lower triangular. No fill-in occurs in the strictly upper part of this block.
3. Compute the updates:
$$A_{12}(= U_{12}) := L_{11}^{-1} A_{12}, \quad (\text{TRSM}) \tag{7}$$
$$A_{22} := A_{22} - L_{21} U_{12}, \quad (\text{GEMM}) \tag{8}$$
$$A_{32} := A_{32} - L_{31} U_{12}. \quad (\text{GEMM}) \tag{9}$$
4. Obtain the copy of the lower triangular part of $A_{13}$, $W_{13} := \text{TRIL}(A_{13})$; compute the updates
$$W_{13}(= U_{13}) := L_{11}^{-1} W_{13}, \quad (\text{TRSM}) \tag{10}$$
$$A_{23} := A_{23} - L_{21} W_{13}, \quad (\text{GEMM}) \tag{11}$$
$$A_{33} := A_{33} - L_{31} W_{13}; \quad (\text{GEMM}) \tag{12}$$
and copy back $A_{13} := \text{TRIL}(W_{13})$.

5. Undo the permutations on $\left[L_{11}^T, L_{21}^T, W_{31}^T\right]^T$ so that these blocks store the multipliers used in the LU factorization in (6) (as corresponds to a LINPACK-style LU factorization) and $W_{31}$ is upper triangular; copy back $A_{31} := \text{TRIU}(W_{31})$.

In our notation, after these operations are carried out, $A_{TL}$ (the part that has been already factorized) grows by $b$ rows/columns so that

$$A = \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & A_{01} & A_{02} & & \\ \hline A_{10} & A_{11} & A_{12} & A_{13} & \\ \hline A_{20} & A_{21} & A_{22} & A_{23} & A_{24} \\ \hline & A_{31} & A_{32} & A_{33} & A_{34} \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right), \tag{13}$$

i.e., $A_{TL} \in \mathbb{R}^{(k+b)\times(k+b)}$, in preparation for the next iteration.

Provided $b \ll k_u, k_l$ (in practice, to optimize cache usage, $b \approx 32$ or 64) the update of $A_{22}$ involves most of the floating-point arithmetic operations (flops). This operation can be cast in terms of the matrix-matrix product, a computation that features an ample degree of parallelism and, therefore, we can expect high performance from GBTRF provided a tuned implementation of GEMM is used.

On the other hand, the algorithm presents also two important drawbacks regarding its implementation in parallel architectures:

- The triangular structure of block $A_{13}/U_{13}$ is not exploited in computations (10)–(12) as there exists no kernel in BLAS to perform such a specialized operation. Consequently, an additional storage is required, as well as two extra copies, and a non-negligible amount of useless flops that involve null elements are performed in these operations.
- Forced by the storage scheme and the lack of specialized BLAS kernels, the updates to be performed during an iteration are split into several small operations with reduced inner parallelism.

### 2.2 Solution of triangular systems

Given the LU factorization computed by GBTRF in (2), routine GBTRS from LAPACK tackles the subsequent band triangular systems to obtain the solution of (1). For this purpose, the routine proceeds as follows:

1. For $i = 0, 1, \ldots, n-2$, (in that strict order,) apply the permutation matrix $P_i$ to the right-hand side term $B$, and update this matrix with the corresponding multipliers in $L_i$:
$$\begin{array}{ll} B := P_i B, & (\text{SWAP}) \\ B := L_i^{-1} B. & (\text{GER}) \end{array} \tag{14}$$
2. For $j = 1, 2, \ldots, m$ solve a triangular system with coefficient matrix $U$ and the right-hand side vector given by the $j$-th column of $B$ (denoted as $B_j$)
$$B_j := U^{-1} B_j. \qquad (\text{TBSV}) \tag{15}$$

Despite the operation tackled by GBTRS belongs, by definition, to the Level-3 BLAS, in this implementation it is entirely cast in terms of less efficient BLAS-2 kernels. This is due to the adoption of the packed storage scheme, the decision of not forming the triangular factor $L$ explicitly (to save storage space), and the lack of a routine in the BLAS specification to solve a triangular system with multiple right-hand sides when the coefficient matrix presents a triangular band structure. We note that routine GETRS, which performs the analogous operation in the non-banded case, does not suffer from these shortcomings.

## 3 New Hybrid CPU-GPU Band Solvers

The algorithm underlying the routine GBTRF invokes, at each iteration, routine TRSM twice and routine GEMM four times (steps 3 and 4). This partitioning of the work is due to the particular storage scheme adopted for band matrices in LAPACK. However, since in LAPACK, concurrency is extracted from the usage of multithreaded implementations of the BLAS kernels, the fragmentation of the computations into small operations potentially limits the performance of the codes. This feature is specially harmful when the algorithm is executed on many-core architectures, like the GPUs, where computations involving large data sets and many flops are mandatory to exploit the capabilities of this type of architectures.

Similarly, routine GBTRS casts its computations in terms of BLAS-2 kernels, e.g. solving for every column of the right-hand side independently. Again, the computations are fragmented, and consequently, their inner concurrency is reduced.

In this section we present GPU-friendly implementations for the routines GBTRF and GBTRS. To adapt their execution to the target platforms, we perform a reordering of the computations and minimum changes in the data storage scheme that permit to merge computations, the use of BLAS-3 kernels (instead of BLAS-2), and improve the inner concurrency of some kernels. Therefore, the new method is more suitable for architectures with a medium to large number of computational units, like current multi-core processors and GPUs. The drawback of the proposal is that the new storage format implies a moderate increment in the memory requirements but, as will be demonstrated later, at the same time it also yields important gains in terms of performance.

### 3.1 Routine GBTRF+M

Assume the packed data structure containing $A$ (see Figure 1-center) is padded with $b$ extra rows at the bottom, with all the entries in this additional space initially set to zero. Then, steps 1–4 in the original implementation of GBTRF can be transformed as follows:

1. In the first step, the LU factorization with partial pivoting

$$P_1 \begin{pmatrix} A_{11} \\ A_{21} \\ A_{31} \end{pmatrix} = \begin{pmatrix} L_{11} \\ L_{21} \\ L_{31} \end{pmatrix} U_{11} \tag{16}$$

   is computed and the blocks of $L$ and $U$ overwrite the corresponding blocks of $A$. There is no longer need for workspace $W_{31}$ nor copies to/from it as the additional rows at the bottom accommodate the elements in the strictly lower triangle of $L_{31}$. Although GBTF2 can be used to complete this computation, it will still require to undo the permutations performed by GBTF2 to keep the upper triangular structure of the block $L_{31}$.
2. Apply the permutations in $P_1$ to the remaining columns of the matrix:

$$\begin{pmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} := P_1 \begin{pmatrix} A_{12} & A_{13} \\ A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} \qquad \text{(LASWP)}. \tag{17}$$

   A single call to LASWP suffices now as the zeros at the bottom of the data structure and the additional $k_l$ superdiagonal set to zero in the structure ensure that fill-in may only occur in the elements in the lower triangular part of $A_{13}$.
3. Compute the updates:

$$(A_{12},\ A_{13})\,(=(U_{12},\ U_{13})) := L_{11}^{-1}\,(A_{12},\ A_{13}) \qquad \text{(TRSM)}, \tag{18}$$

$$\begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} := \begin{pmatrix} A_{22} & A_{23} \\ A_{32} & A_{33} \end{pmatrix} - \begin{pmatrix} L_{21} \\ L_{31} \end{pmatrix} (U_{12},\ U_{13}) \qquad \text{(GEMM)}. \tag{19}$$

   The lower triangular system in (18) returns a lower triangular block in $A_{13}$.
4. Undo the permutations on $\left[L_{11}^T, L_{21}^T, L_{31}^T\right]^T$ so that these blocks store the multipliers used in the LU factorization in (6) and $L_{31}$ is upper triangular.

This reorganization of the algorithm is rich in matrix-matrix products, and hence, it is suitable for massively parallel architectures. The implementation in GBTRF+M takes profit of this enhanced concurrency with the purpose of efficiently exploiting the capabilities of the CPU-GPU platform. In particular, the factorization of the narrow panel (16), which presents a fine-grain parallelism and a modest computational cost, is executed in the CPU. On the other hand, the application of the permutations (17) and the updates (18)–(19) are performed in the GPU, in order to reduce the CPU-GPU communications and exploit the massively parallel architecture of the graphics accelerator.

This implementation executes each operation in the most convenient device while incurring into a moderate number of data transfers between CPU and GPU. In particular, it requires an initialization phase where the matrix $A$ is moved to the GPU before the factorization commences. Then, during the factorization, two copies must be performed per iteration:

1. The entries of $[A_{11}^T, A_{21}^T, A_{31}^T]^T$ are transferred to the GPU after the factorization of this block in (16).
2. The entries that will form $[A_{11}^T, A_{21}^T, A_{31}^T]^T$ during the next iteration are transferred to the CPU after their update as part of (19).

Note that the amount of data transferred at each iteration is moderate in relation to the number of flops, since the number of rows and columns of the block $[A_{11}^T\, A_{21}^T\, A_{31}^T]^T$ are $(k_u + k_l + k_l)$ and $b$, respectively. Furthermore, upon completion of the algorithm, the resulting matrix is replicated in the CPU and GPU, so it can be used during subsequent computations in both devices.

### 3.2 Routine GBTRF+LA

Routine GBTRF+LA is an improved variant of routine GBTRF+M which incorporates look-ahead [10] to further overlap the computations performed by CPU and GPU. Concretely, GBTRF+LA reorders the computations as follows: the updates in (17)–(19), which involve $k_u + k_l$ columns of the matrix, are split column-wise, such that the first $b$ columns are computed first. Then, the updated elements that form the block $[A_{11}^T, A_{21}^T, A_{31}]^T$ of the next iteration ($b$ columns), are sent to the CPU, where the factorization of this block can be performed in parallel with the updates of the remaining $k_u + k_l - b$ columns corresponding to (17)–(19) in the GPU.

This variant requires minimal changes to the codes. Despite it demands the execution in the GPU of kernels with a moderate number of flops, it permits that computations proceed concurrently in both devices, reporting higher performance whenever $b \ll k_u + k_l$.

### 3.3 Routine GBTRS+M

The main drawback of LAPACK routine GBTRS is the absence of BLAS-3 kernels in its implementation. This is due to the adoption of the packed storage format and the lack of the appropiate BLAS routines. Unfortunately, the modifications introduced in the storage scheme still limit the use of BLAS-3 kernels. In particular, as the matrix $L$ is not explicitly formed, in principle the update in (14) must be performed by means of a rank-1 update operation (routine GER from BLAS). However, it is possible to employ BLAS-3 in the solution of the systems in (15). For this purpose, we developed a new routine, named TBSM following the LAPACK convention, that performs this operation via BLAS-3 kernels (mainly matrix-matrix products). Thus, GBTRS presents two parts, the first one is a loop that updates $B$ as in (14). Afterwards, a single call to TBSM solves (15) for all $j$, as described in Section 3.4.

We provide two implementations for this routine, a CPU and a GPU variant. The convenience of these implementations depends on the coefficient matrix dimension ($n$) and in the number of columns of $B$ ($m$). But as usually $m \ll n$, we can focus our analysis on the coefficient matrix dimension. The CPU variant

is suitable for medium to small values of $n$, since it does not require any CPU-GPU data-transfer and the computational cost of the operation is moderate. On the contrary, when $n$ is large, the GPU implementation is more suitable due to the large computational cost and the inner concurrency of the operations involved. Note that the GPU implementation only requires to transfer the matrix $B$ to/from the GPU, as the use of the routines GBTRF+M or GBTRF+LA ensures that the factors $L$ and $U$ are already stored in the GPU memory. Additionally, as all the computations are performed in the GPU, no CPU-GPU synchronizations are required.

### 3.4 Routine TBSM

Consider the row partitioning of the right-hand side matrix $B$, to be overwritten with the solution $X$ to (1),

$$B = \left(\begin{array}{c} B_T \\ \hline B_M \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline B_4 \end{array}\right), \tag{20}$$

where $B_B,\ B_4$ have both $k$ rows (with $k$ an integer multiple of $b$), $B_1,\ B_3$ have $b$ rows each, and $B_2$ has $u = k_u + k_l - b$ rows. Here, $B_B$ represents the part of the right-hand side which has already been overwritten with the corresponding entries of the solution $X$.

Consider also the following conformal partitioning for the upper triangular factor $U$ resulting from the factorization

$$U = \left(\begin{array}{c|c|c} U_{TL} & U_{TM} & \\ \hline & U_{MM} & U_{MR} \\ \hline & & U_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} U_{00} & U_{01} & U_{02} & & \\ \hline & U_{11} & U_{12} & U_{13} & \\ \hline & & U_{22} & U_{23} & U_{24} \\ \hline & & & U_{33} & U_{34} \\ \hline & & & & U_{44} \end{array}\right), \tag{21}$$

where $U_{BR},\ U_{44} \in \mathbb{R}^{k\times k}$; $U_{13}, U_{33} \in \mathbb{R}^{b\times b}$ are lower and upper triangular, respectively; and $U_{22} \in \mathbb{R}^{u\times u}$.

Then, in order to proceed forward, the following operations are required at this iteration:

$$B_3 := U_{33}^{-1} B_3, \qquad \text{(TRSM)} \tag{22}$$

$$B_2 := B_2 - U_{23} B_3, \qquad \text{(GEMM)} \tag{23}$$

$$B_1 := B_1 - U_{13} B_3. \tag{24}$$

The last update involves a triangular matrix ($U_{13}$) and can be performed by means of the BLAS routine TRMM. However, this requires an auxiliary storage,

$W_{13} \in \mathbb{R}^{b \times m}$, since this BLAS kernel only performs a product of the form $M := U_{13}\, M$. Therefore, we perform the next operations for (24):

$$W_{13} := B_3, \tag{25}$$
$$W_{13} := U_{13} W_{13}, \qquad \text{(TRMM)} \tag{26}$$
$$B_1 := B_1 - W_{13}. \tag{27}$$

After these operations are completed, in preparation for the next iteration, the boundaries in $B$ and $U$ are simply shifted as

$$\begin{aligned} B = \begin{pmatrix} B_T \\ B_M \\ B_B \end{pmatrix} &\leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \\ \hline B_3 \\ \hline B_4 \end{array}\right) \\ U = \begin{pmatrix} U_{TL} & U_{TM} & \\ \hline & U_{MM} & U_{MR} \\ \hline & & U_{BR} \end{pmatrix} &\leftarrow \left(\begin{array}{c|c|c|c|c} U_{00} & U_{01} & U_{02} & & \\ \hline & U_{11} & U_{12} & U_{13} & \\ \hline & & U_{22} & U_{23} & U_{24} \\ \hline & & & U_{33} & U_{34} \\ \hline & & & & U_{44} \end{array}\right). \end{aligned} \tag{28}$$

## 4 Experimental Evaluation

In this section we analyze the computational performance of the new routines for the band LU factorization (GBTRF+M and GBTRF+LA) as well as two implementations of the triangular band solvers that follow the algorithm for this phase described in the previous section, but differ in the target architecture: CPU or GPU (denoted as GBTRS+M$_{CPU}$ and GBTRS+M$_{GPU}$ hereafter). Their performances are compared with that of the analogous routines in release 11.1 of Intel MKL (denoted hereafter as GBTRF$_{\text{Intel}}$ and GBTRS$_{\text{Intel}}$).

| Platform | Processors | #cores | Frequency (GHz) | L3 cache (MB) | Memory (GB) |
|---|---|---|---|---|---|
| ENRICO | intel i7-2600 | 4 | 3.4 | 8 | 16 |
| | NVIDIA S2070 | 448 | 1.15 | – | 6 |

**Table 1.** Hardware platform employed in the experimental evaluation.

The performance evaluation was carried out using a hardware platform, ENRICO, equipped with an NVIDIA S2070 ("Fermi") GPU and an Intel four-core

processor; see Table 1. All experiments reported next were performed using IEEE double-precision real arithmetic. We employed band linear systems with 6 different coefficient matrix dimensions $n$ =12,800, 25,600, 38,400, 51,200, 64,000 and 76,800. For each dimension, we generated 3 instances which varied in the bandwidth, $k_b = k_u = k_l$ =1%, 2% and 4% of $n$. We evaluated several algorithmic block sizes ($b$) for each kernel, but for brevity, we only include the results corresponding to the best block size tested.

Table 2 compares the three codes for the band LU factorization: the two new hybrid CPU-GPU implementations, GBTRF+M and GBTRF+LA, and MKL's $\text{GBTRF}_{\text{Intel}}$. These results demonstrate the superior performance of the new implementations when the volume of computations is large. Concretely, both hybrid codes outperform the MKL routine for large matrices while they are still competitive for relatively small matrices. This was expected, as the hybrid routines incur a communication overhead that can be compensated only when the problem is considerably large. In summary, GBTRF+M and GBTRF+LA are superior to MKL for the factorization of matrices with $n$ >25,600 and $k_b$ =2% of $n$. When $n$ >51,200, the new variants are faster than MKL even when $k_b$ =1% of $n$. For the largest problem tested, $n$ =76,800, the acceleration factors obtained by GBTRF+LA with respect to the MKL code are 2.0, 3.9 and 5.5× for $k_b$ =1, 2 and 4% of $n$ respectively. The performance obtained by GBTRF+M is slightly lower, reporting acceleration factors of 1.9, 3.5 and 5.0× for the same problems.

Additionally, we compared MKL's triangular band solver $\text{GBTRS}_{\text{Intel}}$ against both alternative codes proposed in this work (GBTRS+$\text{M}_{CPU}$ and GBTRS+$\text{M}_{GPU}$), on the solution of a linear system with a single right-hand side ($m = 1$). In this scenario, the execution times were comparable though, in general, the performance of the MKL implementation was slightly higher than that of our routines. It is important to note that the new optimizations should be more beneficial when several systems are solved for the same coefficient matrix (i.e., $m > 1$). This is the case in several engineering applications and, in particular, in our target control application. We also remark that MKL is not an open source library, so its implementation may differ from that described in Section 2. In particular, it is likely that MKL also uses BLAS-3 kernels for the triangular band solver, which could explain the similarities between its performance and that of the new implementations.

Considering the results and the reduced impact of GBTRS in the total runtime of the solver, we decided to use the MKL implementation for the solution of the triangular band linear systems. Figure 2 illustrates the speed-up achieved by the best CPU-GPU routine for LU factorization (left) and for the complete band system solver (right). In both cases the reference to compute the acceleration is the solver provided by the MKL library (i.e., routines $\text{GBTRF}_{\text{Intel}}$ and $\text{GBTRS}_{\text{Intel}}$). As most of the flops correspond to the computation of the LU factorization, the speed-ups obtained for the complete solver are similar to those for the LU.

| Matrix Dimension | Bandwidth $k_b = k_u = k_l$ | GBTRF$_{\text{Intel}}$ | GBTRF+M | GBTRF+LA |
|---|---|---|---|---|
| 12,800 | 1% | 0.066 | 0.174 | 0.180 |
| | 2% | 0.142 | 0.240 | 0.245 |
| | 4% | 0.385 | 0.358 | 0.341 |
| 25,600 | 1% | 0.313 | 0.482 | 0.493 |
| | 2% | 0.786 | 0.701 | 0.691 |
| | 4% | 3.397 | 1.339 | 1.231 |
| 38,400 | 1% | 0.684 | 0.867 | 0.844 |
| | 2% | 2.588 | 1.502 | 1.393 |
| | 4% | 11.742 | 3.517 | 3.407 |
| 51,200 | 1% | 1.898 | 1.537 | 1.399 |
| | 2% | 6.989 | 3.131 | 2.496 |
| | 4% | 31.745 | 7.217 | 6.627 |
| 64,000 | 1% | 3.104 | 2.175 | 2.029 |
| | 2% | 12.241 | 4.465 | 4.053 |
| | 4% | 52.701 | 12.796 | 11.660 |
| 76,800 | 1% | 5.749 | 3.044 | 2.808 |
| | 2% | 24.490 | 6.914 | 6.286 |
| | 4% | 103.264 | 20.462 | 18.769 |

**Table 2.** Execution time (in seconds) for the LU factorization of band matrices in ENRICO.

## 5 Application to Model Reduction

In this section we evaluate the impact of the new CPU-GPU banded solvers on the solution of Lyapunov equations of the form

$$AX + XA^T + BB^T = 0, \tag{29}$$

where $A \in \mathbb{R}^{n\times n}$ is sparse, $B \in \mathbb{R}^{n\times m}$, with $m \ll n$, and $X \in \mathbb{R}^{n\times n}$ is the sought-after solution. This linear matrix equation has important applications, among others, in model reduction and linear-quadratic optimal control problems; see, e.g., [11].

The Lyapunov solver employed in our approach consists in a modified version of the *low-rank Cholesky factor - alternating directions implicit* (LRCF-ADI) method [12]. This iterative solver benefits from the frequently encountered low-rank property of the $BB^T$ factor in (29) to deliver a low-rank approximation to a Cholesky or full-rank factor of the solution matrix. Specifically, given an "$l$–cyclic" set of complex shift parameters $\{p_1, p_2, \ldots\}$, $p_k = \alpha_k + \beta_k\, \imath$, with $\imath = \sqrt{-1}$ and $p_k = p_{k+l}$, the cyclic *low-rank alternating directions implicit* (LR-ADI) iteration can be formulated as follows:

$$\begin{aligned} V_0 &= (A + p_1 I_n)^{-1} B, & \hat{S}_0 &= \sqrt{-2\ \alpha_1}\ V_0, \\ V_{k+1} &= V_k - \delta_k (A + p_{k+1} I_n)^{-1} V_k, & \hat{S}_{k+1} &= \left[\hat{S}_k\ ,\ \gamma_k V_{k+1}\right], \end{aligned} \tag{30}$$

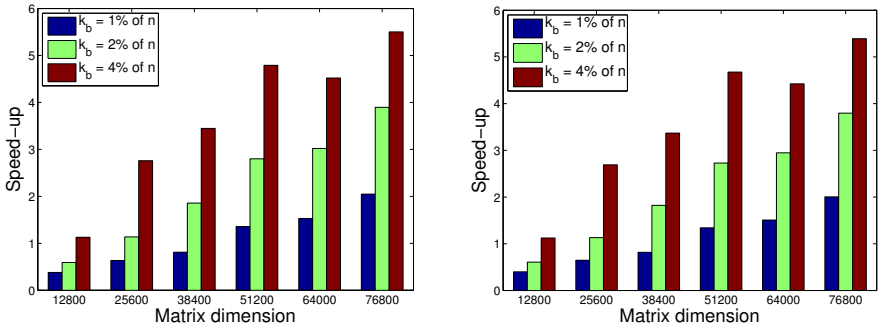


**Fig. 2.** Speed-ups of the new hybrid CPU-GPU solvers against their MKL counterparts for the factorization (left) and the complete solver (right).

where $\gamma_k = \sqrt{\alpha_{k+1}/\alpha_k}$, $\delta_k = p_{k+1} + \overline{p_k}$, with $\overline{p_k}$ the conjugate of $p_k$, and $I_n$ denotes the identity matrix of order $n$. On convergence, after $\hat{k}$ iterations, a low-rank matrix $\hat{S}_{\hat{k}} \in \mathbb{R}^{n \times \hat{k}m}$ is computed such that $\hat{S}_{\hat{k}} \hat{S}_{\hat{k}}^T \approx X$.

It should be observed that the main computations in (30) consist in solving linear systems with multiple ($m$) right-hand sides. Therefore the application of our new solver should significantly accelerate the ADI iteration.

Our approach to tackle the sparse structure of the coefficient matrix $A$ in (29) applies a reordering based on the Reverse Cuthill-McKee algorithm [13] to transform the sparse linear systems in the expressions for $V_0$ and $V_{k+1}$ in (30) into analogous problems with band coefficient matrix. In particular, we evaluated this approach using the Lyapunov equations associated with two instances of the RAIL model reduction problem from the Oberwolfach benchmark collection [14]; see Table 3.

| Problem | $n$ | $k_u = k_l$ | # nonzeros | $m$ |
|---|---|---|---|---|
| $\text{RAIL}_S$ | 5,177 | 139 | 35,185 | 7 |
| $\text{RAIL}_L$ | 20,209 | 276 | 139,233 | 7 |

**Table 3.** Instances of the RAIL example from the Oberwolfach model reduction collection employed in the evaluation.

Table 4 reports the execution times obtained with the Lyapunov MKL-based band solver and the new hybrid CPU-GPU band solver. The results show that the new hybrid Lyapunov solver outperforms its MKL counterpart in both problems, with speed-ups varying between 2.23× for the small instance and 3.14× for the large case.

| Problem | MKL solver | GPU-based solver | Speed-up |
|---|---|---|---|
| $\text{RAIL}_S$ | 2.34 | 1.05 | 2.23 |
| $\text{RAIL}_L$ | 17.71 | 5.65 | 3.14 |

**Table 4.** Execution time (in seconds) and speed-up obtained with the hybrid CPU-GPU variant using Lyapack.

It is also worth to point out that the new solver not only outperforms MKL during the LU factorization phase, but also for the subsequent solution of triangular band linear systems. The reason in this case is that the linear systems in (30) involve $m$ =7 right-hand side vectors which renders the superior performance of the routine GBTRS+$\text{M}_{CPU}$ over MKL for this problem range.

## 6 Concluding Remarks

We have presented new hybrid CPU-GPU routines that accelerate the LU factorization and the subsequent triangular solves for band linear systems, by off-loading the computationally expensive operations to the GPU. Our first CPU-GPU implementation for the LU factorization stage computes the BLAS-3 operation on the hardware accelerator by invoking appropriate kernels from nVIDIA CUBLAS while reducing the amount of CPU-GPU communication. The second GPU variant for this operation incorporates a look-ahead strategy to overlap the update in the GPU with the factorization of the next panel in the CPU.

The experimental results obtained using several band test cases (with dimensions between 12,800 and 76,800 and a bandwidth of 1%, 2% and 4% of the problem size), in a platform equipped with an nVIDIA 2070, reveals speed-ups for the CPU-GPU LU factorization of up to 6×, when compared with the corresponding factorization routine from Intel MKL. The advantages of the hybrid band routines carry over to the solution of sparse Lyapunov solvers, with an acceleration factor around 2-3× with respect to the analogous solver based on MKL.

As part of future work, we plan to enhance the performance of the Lyapunov solver by off-loading to the GPU other band linear algebra operations present in the LR-ADI method as, e.g., the band matrix-vector product. Furthermore, we plan to study the impact of the new GPU-accelerated algorithms on energy consumption.

## Acknowledgements

Ernesto Dufrechou, Pablo Ezzatti and Pablo Igounet acknowledge support from Programa de Desarrollo de las Ciencias Básicas, and Agencia Nacional de Investigación e Innovación, Uruguay. Enrique S. Quintana-Ortí was supported by project TIN2011-23283 of the Ministry of Science and Competitiveness (MINECO) and EU FEDER, and project P1-1B2013-20 of the Fundació Caixa Castelló-Bancaixa and UJI.

## References


1. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
2. Du Croz, J., Mayes, P., Radicati, G.: Factorization of band matrices using level 3 BLAS. LAPACK Working Note 21, Technical Report CS-90-109, University of Tennessee (1990)
3. The Top500 list: Available at `http://www.top500.org` (2013)
4. Kirk, D., Hwu, W.: Programming Massively Parallel Processors, Second Edition: A Hands-on Approach. Morgan Kaufmann (2012)
5. Farber, R.: CUDA application design and development. Morgan Kaufmann (2011)
6. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley (2008)
7. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Euro-Par. (2008) 739–748
8. Benner, P., Ezzatti, P., Quintana-Ortí, E.S., Remón, A.: Matrix inversion on CPU–GPU platforms with applications in control theory. Concurrency and Computation: Practice and Experience **25** (2013) 1170–1182
9. Penzl, T.: LYAPACK: A MATLAB toolbox for large Lyapunov and Riccati equations, model reduction problems, and linear-quadratic optimal control problems. User's guide (version 1.0). Available at `http://www.netlib.org/lyapack/guide.pdf` (2000)
10. 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)
11. Antoulas, A.: Approximation of Large-Scale Dynamical Systems. SIAM Publications, Philadelphia, PA (2005)
12. Penzl, T.: A cyclic low-rank Smith method for large sparse Lyapunov equations. SIAM J. Sci. Comput. **21** (1999) 1401–1418
13. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th National Conference. ACM '69, New York, NY, USA, ACM (1969) 157–172
14. IMTEK: (Oberwolfach model reduction benchmark collection) `http://www.imtek.de/simulation/benchmark/`.