Unleashing GPU Acceleration for Symmetric Band Linear Algebra Kernels and Model Reduction

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

Received: date / Accepted: date

Abstract Linear algebra operations arise in a myriad of scientific and engineering applications and, therefore, their optimization is targeted by a significant number of high performance computing (HPC) research efforts. In particular, the matrix multiplication and the solution of linear systems are two key problems with efficient implementations (or kernels) for a variety of high performance parallel architectures. For these specific problems, leveraging the structure of the associated matrices often leads to remarkable time and memory savings, as is the case, e.g., for symmetric band problems. In this work, we exploit the ample hardware concurrency of many-core graphics processors (GPUs) to accelerate the solution of symmetric positive definite band linear systems, introducing highly tuned versions of the corresponding LAPACK routines. The experimental results with the new GPU kernels reveal important reductions of the execution time when compared with tuned implementations of the same operations provided in Intel's MKL. In addition, we evaluate the performance of the GPU kernels when applied to the solution of model order reduction problems and the associated matrix equations.

P. Benner · A. Remón
Max Planck Institute for Dynamics of Complex Technical Systems, Magdeburg, Germany
E-mail: {benner,remon}@mpi-magdeburg.mpg.de

E. Dufrechou · P. Ezzatti
Instituto de Computación, Universidad de la República, Montevideo, Uruguay
E-mail: {edufrechou,pezzatti}@fing.edu.uy

E. S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaime I, Castellón, Spain
E-mail: quintana@icc.uji.es

1 Introduction

Linear systems with symmetric positive definite (s.p.d.) band coefficient matrix arise, among others, in the numerical solution of partial differential equations, finite element analysis in civil engineering, and as part of matrix equation solvers in control and systems theory. In practice, exploiting the structure of the matrix in these problems yields huge savings, both in the number of computations and storage space. Therefore, it is natural that LAPACK [1,10] comprises efficient methods to solve s.p.d. band linear systems on general-purpose (multicore) processors, provided a tuned (multi-threaded) implementation of BLAS is available.

Hybrid compute servers, consisting of general-purpose multicore processors (CPUs) and a graphics processing unit (GPU), have evolved dramatically in the last decade, becoming an interesting platform to tackle many scientific and engineering applications with high computational requirements [19]. Among the reasons that have contributed to the progressive adoption of GPU hardware accelerators, we can highlight the introduction of application programming interfaces such as CUDA [15,12], OpenCL [14] and OpenACC [16], in conjunction with affordable price, impressive raw performance, and favorable power–performance ratio. In particular, in the area of dense linear algebra, many studies have recently demonstrated the benefits of GPU computing; see, among many others, [20,3,6,5].

In this paper we present new LAPACK-style routines (kernels) that leverage the large hardware concurrency of hybrid CPU-GPU platforms to accelerate the solution of s.p.d. band linear systems. In particular, our experimental evaluation, performed on two platforms equipped with hardware from the two latest generations of NVIDIA's GPUs (NVIDIA "Fermi" S2070 and NVIDIA "Kepler" K20), exposes that the GPU-enabled routines offer superior performance and scalability as compared to the highly-tuned multithreaded symmetric band kernels in Intel's MKL (Math Kernel Library). Furthermore, the application of the novel codes to a matrix equation solver shows how 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 revisit the BLAS routines for the band symmetric matrix multiplication and the GPU versions proposed in [11]. Next, in Section 3, we study the LAPACK strategies for the solution of s.p.d. symmetric band linear systems on multicore architectures, and we introduce our new hybrid CPU-GPU routines in detail. Section 4 presents the experimental evaluation of our new solvers, and Section 5 analyses the application of the new kernels to the solution of band symmetric Lyapunov matrix equations arising in model reduction. Finally, a few concluding remarks close the paper in Section 6.

2 Symmetric Band Matrix Multiplication

In this section we describe several kernels that leverage the computational power of GPUs to accelerate the computation of the matrix multiplication:

$$C = AB + C, \tag{1}$$

where A is a symmetric band matrix. In this operation, for simplicity we consider that A, B, C are all three of dimension $n \times n$ hereafter.

2.1 The operation in BLAS

The BLAS specification contains several routines to operate with symmetric band matrices. In particular, BLAS include kernel SBMV to compute a matrix-vector product involving a band symmetric matrix but, in contrast, the interface does not offer the equivalent kernel for matrix multiplication when one of the matrices presents a band structure. To tackle this, the matrix multiplication can be easily implemented on top of the SBMV routine. Concretely, in (1) we can partition the matrix B column-wise, and perform the sequence of matrix-vector products:

$$C_j = AB_j + C_j,\ 1 \leq j \leq n, \tag{2}$$

where C_j, B_j stand for the j-th columns of B and C respectively.

Although this simple approach allows to compute the matrix multiplication using only BLAS kernels, it is based on a level-2 BLAS routine, while the product of matrices is, in principle, a level-3 BLAS operation. Thus, with this implementation each element of the matrix A is accessed n times, in general resulting in a suboptimal usage of the memory hierarchy.

2.2 Algorithm SBMM$_{BLK}$

Blocked algorithms for linear algebra operations take advantage of the memory hierarchy of current computer architectures to hide the limited memory latency and bandwidth and deliver higher performance. In this context, in [11] we proposed the blocked algorithm to compute the matrix multiplication (1) given in Figures 1 and 2. This implementation only accesses the elements in the lower triangular part of A, adhering to the packed storage format employed by BLAS and LAPACK for this type of matrices. Analogous procedures that only access the elements in the upper triangle or all the elements of A are straight-forward.

The algorithm consists of two loops. The outer loop (Figure 1) partitions the matrices B and C into blocks of c columns and, at each iteration, updates the elements in the active column-block of C. The inner loop (Figure 2) proceeds along the main diagonal of A (i.e., from the top-left corner to the bottom-right one), updating the corresponding elements of C. Matrices B and C are partitioned row-wise, while A is partitioned into 3×3 blocks. At each iteration, the blocks A_{i1} and B_i, with $i = 1, 2, 3$, are accessed; while the blocks C_i, are updated. Note that A_{11} and A_{31} are, respectively, lower and upper triangular blocks. Figure 3 details the blocks accessed and updated at a given iteration of the inner loop.

Algorithm SBMM$_{BLK}$ can be conveniently adapted to the underlying architecture and problem by carefully choosing the blocking parameters c and b, which strongly depend on the memory organization of the target architecture and the bandwidth k of the matrix A. For example, in current multicore processors, a "small" b is usually a convenient choice (e.g., $b = 32$), while for GPUs, larger values are recommended (e.g., $b = 128$), see [4,17].

2.3 GPU implementations

We next describe two routines to compute the symmetric band matrix multiplication on a GPU accelerator. These implementations intensively invoke kernels from CUBLAS to carry out the computation. In both cases, the matrices are initially sent to the GPU; the corre-

```
Algorithm: $[C] := \text{SBMM}_{BLK\_outer}(C, A, B, k)$
Partition $C \rightarrow \left( C_L \,\|\, C_R \right)$, $B \rightarrow \left( B_L \,\|\, B_R \right)$
    where $C_L$, $B_L$ have 0 columns

while $n(C_L) < n(C)$ do
    Determine block size $c$
    Repartition
        $\left( C_L \,\|\, C_R \right) \rightarrow \left( C_0 \,\|\, C_1 \,|\, C_2 \right)$, $\left( B_L \,\|\, B_R \right) \rightarrow \left( B_0 \,\|\, B_1 \,|\, B_2 \right)$
            where $C_1$, $B_1$ have $c$ columns
    ========================================
    $C_1 := \text{SBMM}_{BLK\_inner}(C_1, A, B_1, k)$
    ========================================
    Continue with
        $\left( C_L \,\|\, C_R \right) \leftarrow \left( C_0 \,|\, C_1 \,\|\, C_2 \right)$, $\left( B_L \,\|\, B_R \right) \leftarrow \left( B_0 \,|\, B_1 \,\|\, B_2 \right)$
endwhile
```

Fig. 1 Outer loop of Algorithm SBMM_{BLK} that computes $C := AB + C$. In the notation, $n(\cdot)$ returns the number of columns of a matrix.

```
Algorithm: $[E] := \text{SBMM}_{BLK\_inner}(E, A, D, k)$
Partition $E \rightarrow \left(\begin{array}{c} E_T \\ \hline E_M \\ \hline E_B \end{array}\right)$, $A \rightarrow \left(\begin{array}{c|c} A_{TL} & \\ \hline A_{ML} & A_{MR} \\ \hline & A_{BR} \end{array}\right)$, $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_M \\ \hline D_B \end{array}\right)$
    where $E_T$, $D_T$ have 0 elements; $A_{TL}$ is $0 \times 0$ and $E_M$, $A_{ML}$ have $k$ rows
while $m(E_T) < m(E)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c} E_T \\ \hline E_M \\ \hline E_B \end{array}\right) \rightarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ E_2 \\ E_3 \\ \hline E_4 \end{array}\right)$, $\left(\begin{array}{c|c} A_{TL} & \\ \hline A_{ML} & A_{MR} \\ \hline & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|cc} A_{00} & & \\ \hline A_{10} & A_{11} & \\ A_{20} & A_{21} & A_{22} \\ \hline & A_{31} & A_{32} \\ & & A_{42} \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_M \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ D_2 \\ D_3 \\ \hline D_4 \end{array}\right)$
        where  $D_1$ has $b$ rows;
               $E_1$ has $b$ rows;
               $E_3$ has 0 rows if $m(D_0) > (n(A) - k - 1)$ and has $b$ rows otherwise;
               $A_{11}$ is $b \times b$;
               $A_{31}$ is empty if $m(D_0) > (n(A) - k - 1)$ and is $b \times b$ otherwise;
    ========================================
    $E_1 := E_1 + A_{11} D_1$
    $E_1 := E_1 + A_{21}^T D_2$
    $E_1 := E_1 + A_{31}^T D_3$
    $E_2 := E_2 + A_{21} D_1$
    $E_3 := E_3 + A_{31} D_1$
    ========================================
    Continue with
        $\left(\begin{array}{c} E_T \\ \hline E_M \\ \hline E_B \end{array}\right) \leftarrow \left(\begin{array}{c} E_0 \\ E_1 \\ \hline E_2 \\ \hline E_3 \\ E_4 \end{array}\right)$, $\left(\begin{array}{c|c} A_{TL} & \\ \hline A_{ML} & A_{MR} \\ \hline & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{cc|c} A_{00} & & \\ A_{10} & A_{11} & \\ \hline A_{20} & A_{21} & A_{22} \\ & A_{31} & A_{32} \\ \hline & & A_{42} \end{array}\right)$, $\left(\begin{array}{c} D_T \\ \hline D_M \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ D_1 \\ \hline D_2 \\ \hline D_3 \\ D_4 \end{array}\right)$
endwhile
```

Fig. 2 Inner loop of Algorithm SBMM_{BLK} that computes $C := AB + C$. In the notation, $m(\cdot)$ returns the number of rows of a matrix.

sponding operations are next executed; and the result is finally transferred back to the CPU.

2.3.1 Kernel SBMM_{blk}

Routine SBMM_{blk} is an implementation of Algorithm SBMM_{BLK} with all the computations performed via the appropriate CUBLAS kernels. The update of C_1 re-

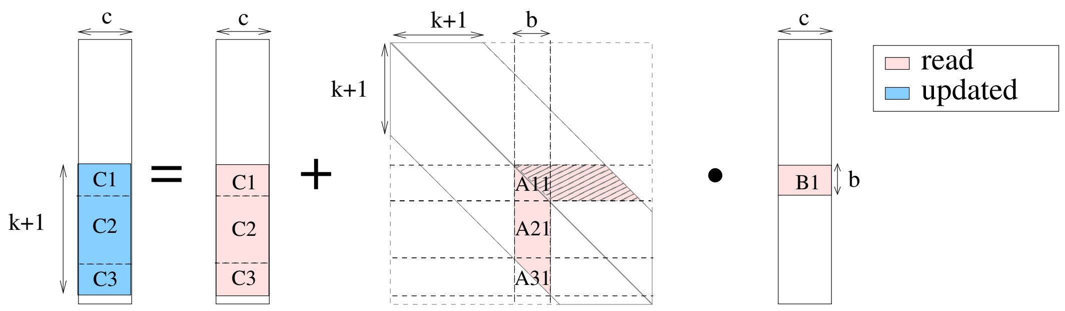


Fig. 3 Elements read and updated during an iteration of the inner loop of the SBMM$_{BLK}$ algorithm. The matrix bandwidth is denoted as k.

quires three matrix multiplications: one involving a symmetric block (A_{11}); a second with an upper triangular block (A_{31}); and the last one with two general matrices. CUBLAS provides specific routines for all these operations. In addition, C_2 and C_3 are respectively updated via a product of two general matrices and a product of an upper triangular matrix times a general matrix.

The use of CUBLAS routines also presents a drawback, as the kernel that implements the product of a triangular matrix times a general matrix (routine TRMM) indeed computes

$$C = op(A)\ B, \tag{3}$$

where A is an upper or lower triangular matrix and $op(A)$ denotes A or A^T. In contrast, the updates of C_1 and C_3 both require an operation of the type

$$C = C + op(A)\ B, \tag{4}$$

To overcome this problem, routine SBMM$_{blk}$ updates C_1 with the following sequence of operations:

(TRMM) $W = A_{31}^T B_1$,
(GEAM) $C_1 = C_1 + W$.

(Next to each operation, we indicate the CUBLAS kernel that implements it.) The update of C_3 is analogous. In both cases, this procedure requires an auxiliary workspace W of dimension $b \times c$.

High performance can be expected of this implementation due to the use of tuned CUBLAS routines.

2.3.2 Kernel SBMM$_{blk+ms}$

The SBMM$_{blk}$ implementation presents some drawbacks with a negative impact on performance. In particular, it requires up to 6 operations per iteration. Furthermore, two of the operations involve a triangular matrix and are computed in two steps (as discussed in the previous subsection). Thus, up to 8 kernels are invoked at each iteration, and some of them require a low computational effort (e.g., the two invocations to GEAM). The SBMM$_{blk+ms}$ implementation aims to reduce the number of routine invocations by removing those calls with a lower computational cost. To make this possible, we perform some changes in the matrix storage. Concretely, consider that the lower triangular part of the symmetric band matrix A is stored following the BLAS specific format. Then, we add b additional rows to the bottom of A where, for a GPU accelerator, this specific value of b is chosen to enable a coalesced access to the elements of A. (When the upper part of A is stored, then the new rows should be added at the top of A.)

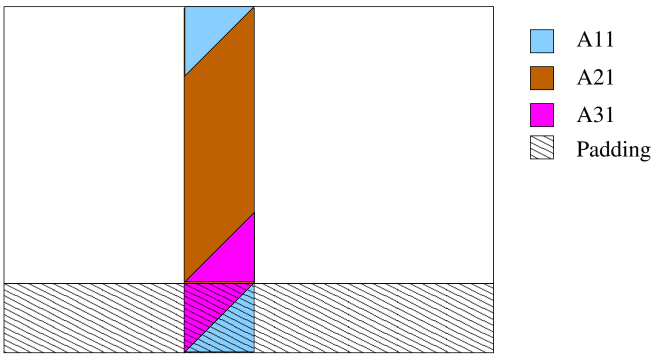


Fig. 4 Modified storage scheme for symmetric band matrices and how it is accessed in the inner loop from SBMM$_{BLK}$.

Figure 4 shows the modified storage scheme and how it is accessed during an iteration of the inner loop of SBMM$_{BLK}$. The strictly lower triangular part of A_{31} is now conveniently placed in the added rows. Consequently, the blocks A_{21} and A_{31} can be merged, and the operations they are involved in can be fused. Thus, the updates performed at each step of the inner loop can be reorganized as:

$$E_1 := E_1 + A_{11} D_1,$$
$$E_1 := E_1 + \left[A_{21}^T A_{31}^T\right] \begin{bmatrix} D_2 \\ D_3 \end{bmatrix},$$
$$\begin{bmatrix} E_2 \\ E_3 \end{bmatrix} := \begin{bmatrix} E_2 \\ E_3 \end{bmatrix} + \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} D_1.$$

This approach presents two major advantages:

– The number of invocations to CUBLAS kernels is reduced from 8 to 3 per step and, consequently, the overhead introduced by the invocations is also reduced.
– It eliminates the calls to kernels with a moderate to low cost, which can not exploit the massively parallel architecture of the GPU. Concretely, the operations that disappear involve triangular matrices and present load-balancing problems.

There are also two drawbacks in this implementation: the memory requirement is enlarged; and the number of arithmetic operations is also increased, as the new kernel operates with the null elements in A_{11} and A_{31}.

3 Solution of S.P.D. Band Linear Systems

In this section, we review the procedure for the solution of s.p.d. band linear systems in LAPACK, and detail how to accelerate this computation when the target is a hybrid CPU-GPU platform.

3.1 LAPACK solver

LAPACK supports the solution of linear systems of the form

$$A\,X = B, \tag{5}$$

where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, and $X \in \mathbb{R}^{n\times m}$ is the sought-after solution. When A is s.p.d., the system is solved in three steps:

1. Compute the Cholesky factorization $A = LL^T$, where L is lower triangular[1].
2. Solve the lower triangular system $L\,Y = B$
3. Solve the upper triangular system $L^T\;X = Y$

BLAS and LAPACK support this method by providing tailored triangular solvers for steps 2–3 and the factorization in step 1, respectively. When A exhibits a band structure, the factorization of this matrix is initially computed using the kernel PBTRF, and then two band triangular systems (involving L and its transpose) are solved for each column of Y, X (routine TBSV). This is necessary since there is no routine in BLAS to solve a linear system with multiple right-hand sides when the coefficient matrix presents a triangular band form.

This implementation benefits from the intensive use of tuned BLAS kernels, but has also some important limitations. Concretely, an important drawback lies in that L is fully accessed $2m$ times, yielding an inefficient use of the memory hierarchy.

3.2 Hybrid CPU-GPU s.p.d. band solvers

3.2.1 Basic solver (GPU$_{V1}$)

This blocked algorithm, illustrated in Figure 5, off-loads the computationally-intensive operations of the factorization to the accelerator, while exploiting the multi-core processor to efficiently execute the fine-grain operations. In particular, the CPU computes the factorization of block A_{11} and the GPU performs the remaining updates. Additionally, we leverage a modified storage scheme analogous to that described in Section 2 in order to reduce the number of operations performed by the GPU at each iteration from 5 to only 2: a triangular solver to update $[A_{21}; A_{31}]$; and a rank-k update for the remaining three blocks.

Once the factorization is computed, the subsequent triangular-band linear systems are solved in the CPU. The reason is twofold: *(i)* usually the number of columns of B is reduced and therefore this operation presents a moderate cost; and *(ii)* its degree of data-parallelism is limited by the inherent dependencies.

In summary, the factorization is accelerated on the GPU via the fusion of operations. Moreover, the triangular solves are accelerated when B presents several columns, as the number of memory accesses to L is reduced to only once per solve.

3.2.2 Merged solver (GPU$_{V2}$)

The three main operations (factorization and two triangular solvers) in the previous variant are computed sequentially. However, the factorization and the first of the triangular band solves can be obtained concurrently, as the first solve requires the elements of L in the same order as they are computed during the factorization. Consequently, as soon as one element/block of L is computed, the corresponding operations of the first triangular band linear solve can be performed. This is especially appealing in our implementation, as the bulk

[1] Alternatively, one can decompose the matrix as $A = U^T U$, where $U = L^T$ is upper triangular.

```
Algorithm: $[A] := \textsc{pbtrf}_{BLK}(A, k)$
Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & A_{TR} \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline A_{BL} & A_{BM} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$, $A_{MM}$ is $k \times k$
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c|c} A_{TL} & \star & \star \\ \hline A_{ML} & A_{MM} & \star \\ \hline A_{BL} & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline A_{10} & A_{11} & \star & \star & \\ A_{20} & A_{21} & A_{22} & \star & \star \\ \hline & A_{31} & A_{32} & A_{33} & \star \\ & & A_{42} & A_{43} & A_{44} \end{array}\right)$
        where $A_{11}, A_{33}$ are $b \times b$, $A_{22}$ is $l \times l$,
              with $l := \min(k - b, m(A) - m(A_{TL}) - b)$
    ---------------------------------------------------------------
    $Copy\_to\_CPU(A_{11})$
    $A_{11} := \textsc{pbtrf}_{UNB}(A_{11})$            $(in\ CPU)$
    $Copy\_to\_GPU(A_{11})$
    $A_{21} := A_{21}\ \textsc{tril}(A_{11})^{-T}$       $(in\ GPU)$
    $A_{31} := A_{31}\ \textsc{tril}(A_{11})^{-T}$       $(in\ GPU)$
    $A_{22} := A_{22} - A_{21}A_{21}^T$            $(in\ GPU)$
    $A_{32} := A_{32} - A_{31}A_{21}^T$            $(in\ GPU)$
    $A_{33} := A_{33} - A_{31}A_{31}^T$            $(in\ GPU)$
    ---------------------------------------------------------------
    Continue with
        $\left(\begin{array}{c|c|c} A_{TL} & \star & \star \\ \hline A_{ML} & A_{MM} & \star \\ \hline A_{BL} & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ A_{10} & A_{11} & \star & \star & \\ \hline A_{20} & A_{21} & A_{22} & \star & \star \\ & A_{31} & A_{32} & A_{33} & \star \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$
endwhile
```

Fig. 5 Algorithm $\textsc{pbtrf}_{BLK}$ that computes the factorization $A = LL^T$. In the notation, $\textsc{pbtrf}_{UNB}(\cdot)$ is an unblocked version of this routine and $\textsc{tril}(\cdot)$ returns the lower triangular part of a matrix.

of the computations during the factorization are performed in the GPU while the system solve is executed in the CPU.

Figure 6 details how the computations involved in the factorization and the first system solve are reorganized in this variant. The new order allows to overlap concurrent computations in both architectures and, in an ideal scenario, completely hides the cost of the first triangular band system solve. This option may also yield a better exploitation of the memory system, due to the use of the elements of L immediately after they are computed, which may incur a lower number volume of cache misses.

	Buenaventura	Enrico
GPU	nVIDIA K20	nVIDIA S2070
#CUDA cores	2,496	448
Frequency (GHz)	0.71	1.15
GPU memory (GB)	6	5
CPU	Intel i3-3220	Intel i7-2600
#CPU cores	2	4
Frequency (GHz)	3.4	3.3
Memory (GB)	16	8
Operating System	CentOS 6.4	CentOS 6.2
C/Fortran	gcc v4.4.7	gcc v4.4.6
CUDA/CUBLAS	5.0	5.5

Table 1 Hardware and software employed.

4 Experimental Evaluation

In this section we analyse the computational performance of the new routines for the solution of s.p.d. band linear systems, GPU_{V1} and GPU_{V2}, comparing them against analogous routines in release 11.1 of Intel's MKL (multi-threaded version).

The evaluation was carried out on two servers, Buenaventura and Enrico, equipped with state-of-the-art nVIDIA GPUs and Intel multi-core processors; see Table 1. All experiments were performed using IEEE double-precision real arithmetic. We generated s.p.d. band coefficient matrices of 8 dimensions $n = 38,400$,

```
Algorithm: $[A, B] := \text{PBTRF\_SOLVER}(A, k, B)$

Partition $A \rightarrow \left(\begin{array}{c|c|c} A_{TL} & A_{TM} & A_{TR} \\ \hline A_{ML} & A_{MM} & A_{MR} \\ \hline A_{BL} & A_{BM} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$
    where $A_{TL}$ is $0 \times 0$, $A_{MM}$ is $k \times k$, $B_T$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
    Determine block size $b$
    Repartition
        $\left(\begin{array}{c|c|c} A_{TL} & \star & \star \\ \hline A_{ML} & A_{MM} & \star \\ \hline A_{BL} & A_{BM} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline A_{10} & A_{11} & \star & \star & \\ \hline A_{20} & A_{21} & A_{22} & \star & \star \\ \hline & A_{31} & A_{32} & A_{33} & \star \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$, $\left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \\ \hline B_3 \end{array}\right)$
        where $A_{11}, A_{33}$ are $b \times b$, $A_{22}$ is $l \times l$,
              $B_1$ is $b \times n(B)$, $B_2$ is $l \times n(B)$,
              with $l := \min(k - b, m(A) - m(A_{TL}) - b)$,
    ========================================================
    $Copy\_to\_CPU(A_{11})$
    $A_{11} := \text{PBTRF}_{UNB}(A_{11})$                                  $(in\ CPU)$
    $Copy\_to\_GPU(A_{11})$
    $\begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} := \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} \text{TRIL}(A_{11})^{-T}$                    $(in\ GPU)$
    $B_1 := B_1 \ \text{TRIL}(A_{11})^{-T}$                                 $(in\ CPU)$
    $Copy\_to\_CPU([A_{21}; A_{31}])$
    $\begin{bmatrix} A_{11} & \star \\ A_{21} & A_{22} \end{bmatrix} := \begin{bmatrix} A_{22} & \star \\ A_{23} & A_{33} \end{bmatrix} - \begin{bmatrix} A_{21} \\ A_{23} \end{bmatrix} [\, A_{21}^T A_{23}^T \,]$   $(in\ GPU)$
    $B_2 := B_2 - \begin{bmatrix} A_{21} \\ A_{31} \end{bmatrix} B_1$                          $(in\ CPU)$
    ========================================================
    Continue with
        $\left(\begin{array}{c|c|c} A_{TL} & \star & \star \\ \hline A_{ML} & A_{MM} & \star \\ \hline A_{BL} & A_{BM} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c|c|c} A_{00} & \star & \star & & \\ \hline A_{10} & A_{11} & \star & \star & \\ \hline A_{20} & A_{21} & A_{22} & \star & \star \\ \hline & A_{31} & A_{32} & A_{33} & \star \\ \hline & & A_{42} & A_{43} & A_{44} \end{array}\right)$
endwhile
```

Fig. 6 Algorithm PBTRF_SOLVER that computes the factorization $A = LL^T$ and simultaneously solves $L \cdot Y = B$, overwriting B with the result Y.

51,200, 64,000, 76,800, 89,600, 102,400, 115,200 and 128,000. For each matrix size, we produced 3 problem instances which varied in bandwidth, k =1%, 2% and 4% of n, except for the two largest dimensions in which the GPU memory could not allocate the problem instance with largest bandwidth. Matrices B and X were selected to have a single column (i.e., $m = 1$ and they are column vectors), so the impact of the solver with multiple right-hand sides is not analyzed. We evaluated several values for the algorithmic blocking parameter b but, for brevity, we only include the results corresponding to the best block size.

Table 2 and Figure 7 compare the performance of the three kernels for band s.p.d. linear systems: the two GPU-accelerated kernels (GPU$_{V1}$ and GPU$_{V2}$) and the MKL implementation on both platforms. For GPU$_{V1}$ and MKL, the costs of the factorization and the solution of triangular systems are decoupled into two columns of Table 2: "Fact." and "Tr. Sol.", respectively. For GPU$_{V2}$, the time includes the factorization and both triangular band solves.

The experimental results demonstrate the superiority of the new routines over the MKL solver, especially when the computational cost is large. In particular, both GPU$_{V1}$ and GPU$_{V2}$ outperform the MKL implementation for all the problem instances in platform BUENAVENTURA. In contrast, in ENRICO the MKL implementation is faster for those cases presenting a moderate computational cost. Specifically, the GPU-based kernels outperform the MKL variant when k = 4% of n for all tested dimensions, and when n >76,800 and k = 2% of n in this platform.

These results were expected considering the difference between the peak performance of the CPU and the GPU that form each platform. Concretely, the CPU in ENRICO has a higher peak performance than that in BUENAVENTURA. However, the K20 GPU in BUENAVENTURA contains a larger number of CUDA cores than the S2070 GPU in ENRICO. In consequence, the gap between the peak performance of the CPU and the GPU in BUENAVENTURA is larger than that for the processors in ENRICO; and BUENAVENTURA is more suited for the GPU-based solvers.

This gap between the CPU and the GPU computational power is especially important for GPU$_{V2}$. This variant off-loads most of the computations during the factorization to the GPU, while the CPU solves a triangular system. Obviously, this technique suits platforms where the GPU is much more powerful than the CPU. As a consequence, GPU$_{V1}$ outperforms GPU$_{V2}$ in ENRICO, but in platform BUENAVENTURA it is the other way around.

On the other hand, B has a single column, which motivates that the MKL triangular solver outperforms the new codes. A separate evaluation of both solvers showed that the cost of the MKL triangular solver increases linearly with the number of columns in B (as expected), while the computational time of the new routine reports only minimal increments when a moderate number of columns (e.g., 10) is added to B. Thus, we suggest to use our routine whenever the number of columns of B is at least 2.

In general, both hybrid codes outperform the MKL routine for large matrices while they are still competitive for relatively small ones. This is explained by the fact that the hybrid routines incur in a communication overhead that can be compensated only when the problem is considerably large.

5 Application in Model Reduction

Several important problems in the area of control, such as model order reduction and linear-quadratic (LQ) optimal control, involve the solution of linear and quadratic matrix equations [2]. In this particular work, we focus our effort on the Lyapunov equation

$$AX + XA^T + BB^T = 0, \qquad (6)$$

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. When n is large, these equations represent the major computational challenge for the solution of model order reduction and LQ optimal control problems.

5.1 The LRCF-ADI method

The *low-rank Cholesky factor–alternating directions implicit* (LRCF-ADI) method [18] is an efficient approach to tackle (6) when the coefficient matrix A is large and sparse or structured (e.g. a symmetric band matrix). This iterative solver benefits from the frequently encountered low-rank property of the BB^T factor in (6) to deliver a low-rank approximation to a Cholesky or full-rank factor of X. 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 shown in Algorithm 1.

Algorithm 1: LR-ADI

```
begin
    $V_0 \leftarrow (A + p_1 I_n)^{-1} B$,
    $\hat{S}_0 \leftarrow \sqrt{-2\ \alpha_1}\ V_0$,
    $k \leftarrow 0$
    repeat
        $V_{k+1} \leftarrow V_k - \delta_k (A + p_{k+1} I_n)^{-1} V_k$,
        $\hat{S}_{k+1} \leftarrow \left[\hat{S}_k\ ,\ \gamma_k V_{k+1}\right]$,
        $k \leftarrow k + 1$
    until until convergence;
```

There, $\gamma_k = \sqrt{\alpha_{k+1}/\alpha_k}$, $\delta_k = p_{k+1} + \overline{p_k}$, 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 $\hat{S}_{\hat{k}} \hat{S}_{\hat{k}}^T \approx X$.

From a computational viewpoint, the performance of the algorithm is determined by that of the matrix multiplication and the sequence of shifted linear systems of the form $(A + p_k I_n)^{-1} X = Y$, where we note that, in practice, Y presents only few columns. Moreover, the algorithm used to calculate the shift parameters p_k is an Arnoldi iteration that relies on the solution of linear systems of the form $Ax = y$, where x and y are both vectors. Hence, it is interesting to evaluate the impact of the new symmetric band solvers on this method.

Note that this variant does not incorporate recent improvements in the ADI method avoiding complex arithmetic and automating the shift selection process [8]. Here, we use the publicly available implementations in Lyapack [7].

5.2 Implementation on hybrid CPU-GPU platforms

In Section 3 we presented two hybrid solvers for s.p.d. band linear systems: GPU$_{V1}$ exploits the GPU to accelerate the computation of the Cholesky factorization, and computes the subsequent triangular band systems in the CPU. GPU$_{V2}$ reorders the operations such that the factorization and the first band triangular solve are

Platform	Dimension	Bandwidth	MKL			GPU$_{V1}$			GPU$_{V2}$
	n	k	Total	Fact.	Tr. Sol.	Total	Fact.	Tr. Sol.	
Buenaventura	38,400	1%	0.24	0.21	0.02	0.23	0.16	0.07	**0.22**
		2%	0.85	0.80	0.04	0.33	0.19	0.13	**0.31**
		4%	3.89	3.79	0.09	0.65	0.39	0.26	**0.62**
	51,200	1%	0.57	0.52	0.04	0.34	0.21	0.12	**0.32**
		2%	2.63	2.55	0.08	0.57	0.33	0.24	**0.53**
		4%	15.67	15.50	0.17	1.21	0.75	0.46	**1.14**
	64,000	1%	0.92	0.85	0.06	0.49	0.30	0.19	**0.47**
		2%	4.37	4.24	0.13	0.92	0.55	0.37	**0.87**
		4%	17.16	16.89	0.27	2.03	1.30	0.72	**1.91**
	76,800	1%	1.68	1.59	0.09	0.66	0.39	0.27	**0.63**
		2%	7.90	7.70	0.19	1.32	0.79	0.52	**1.24**
		4%	36.67	36.27	0.39	3.15	2.09	1.05	**2.94**
	89,600	1%	3.92	3.79	0.13	0.89	0.53	0.36	**0.85**
		2%	11.75	11.49	0.26	1.82	1.11	0.71	**1.73**
		4%	46.99	46.44	0.54	4.68	3.23	1.44	**4.52**
	102,400	1%	5.32	5.13	0.19	1.14	0.66	0.47	**1.08**
		2%	31.44	31.08	0.35	2.44	1.50	0.93	**2.31**
		4%	96.16	95.43	0.73	7.80	5.67	2.12	**6.11**
	115,200	1%	6.47	6.23	0.23	1.44	0.84	0.60	**1.36**
		2%	24.60	24.17	0.43	3.22	2.04	1.18	**3.04**
	128,000	1%	8.88	8.62	0.26	1.90	1.12	0.78	**1.76**
		2%	35.86	35.30	0.56	4.25	2.76	1.48	**3.99**
Enrico	38,400	1%	**0.14**	0.12	0.01	0.23	0.19	0.04	0.23
		2%	**0.40**	0.37	0.03	0.50	0.43	0.07	0.51
		4%	1.79	1.72	0.06	**0.81**	0.67	0.13	**0.81**
	51,200	1%	**0.31**	0.28	0.03	0.43	0.35	0.07	0.42
		2%	**0.98**	0.92	0.05	1.04	0.92	0.12	1.07
		4%	6.02	5.91	0.11	**1.56**	1.32	0.24	1.60
	64,000	1%	**0.49**	0.45	0.04	0.66	0.55	0.11	0.68
		2%	**1.68**	1.59	0.08	1.81	1.61	0.19	1.84
		4%	11.07	10.89	0.17	**2.68**	2.30	0.38	2.69
	76,800	1%	**0.81**	0.75	0.06	1.01	0.86	0.14	1.03
		2%	3.64	3.51	0.12	**1.59**	1.30	0.28	1.64
		4%	19.50	19.25	0.25	4.25	3.70	0.54	**4.24**
	89,600	1%	**1.19**	1.10	0.08	1.43	1.23	0.19	1.46
		2%	6.62	6.46	0.16	**2.24**	1.86	0.37	2.29
		4%	30.40	30.06	0.34	6.29	5.56	0.73	**6.27**
	102,400	1%	**1.97**	1.85	0.11	2.10	1.85	0.25	2.15
		2%	12.23	12.00	0.22	**3.15**	2.65	0.49	3.23
		4%	74.84	74.39	0.44	8.99	8.03	0.96	**8.95**
	115,200	1%	**2.37**	2.23	0.14	2.79	2.48	0.31	2.85
		2%	16.02	15.75	0.27	**4.15**	3.53	0.61	4.21
	128,000	1%	**3.40**	3.23	0.17	3.63	3.24	0.38	3.70
		2%	22.49	22.13	0.35	**5.39**	4.63	0.76	5.44

Table 2 Execution time (in seconds) of the two versions of the hybrid CPU+GPU s.p.d. band system solver and Intel's MKL implementation.

performed concurrently using both architectures, and the second band triangular solver completes the process in the CPU.

In our implementation, we applied GPU$_{V1}$ to calculate the shift parameters, as a single matrix factorization is required for this process. On the other hand, a new factorization is computed per LR-ADI iteration, and we leverage GPU$_{V2}$ for this purpose. We note that only l different coefficient matrices appear during the LR-ADI procedure while, in general, the number of iterations for convergence is larger. Here, due to memory restrictions, we decided to recompute the factorizations instead of storing the Cholesky factors.

Problem	n	k	m
RAIL$_S$	5,177	139	7
RAIL$_M$	20,209	276	7
RAIL$_L$	79,841	550	7

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

Problem	Solver		Speed-up
	MKL	GPU-based	
RAIL$_S$	0.45	0.70	0.64
RAIL$_M$	2.79	2.81	0.99
RAIL$_L$	30.51	15.04	2.02

Table 4 Execution time (in seconds) of the hybrid CPU-GPU Lyapunov solvers and the MKL-based counterpart in BUENAVENTURA, and speed-ups of the GPU-accelerated routines with respect to the CPU-only (MKL) implementation.

5.3 Experimental evaluation

We employed three instances of the RAIL model reduction problem from the Oberwolfach benchmark collection [13] for the evaluation of the Lyapunov solvers. The coefficient matrix A in (6) for these problems is s.p.d. and sparse. Therefore, we applied the Reverse Cuthill-McKee algorithm [9] to transform the unstructured sparse linear systems in the expressions for V_0 and V_{k+1} in Algorithm 1 into s.p.d. problems with band coefficient matrix; see Table 3 where n is the matrix dimension, k is the resulting bandwidth and m the number of columns of the rigth hand side matrix (i.e. B).

Table 4 reports the execution times obtained from the Lyapunov solvers that employ Intel's MKL to perform the computations in the CPU compared with our hybrid CPU-GPU solvers (GPU$_{V1}$-GPU$_{V2}$) in BUENAVENTURA. The results show that the hybrid Lyapunov solver outperforms its MKL counterpart for the largest problem instance, reaching a speed-up factor around 2×. For the medium-size problem, the performance of both solvers is similar; and for the smallest one, the MKL-based solver is the best option. This was expected as large problems are needed to exploit the capabilities of the GPU and, therefore, compensate the overhead incurred by the CPU-GPU data transfers. It should be noted that this result is strongly aligned with the obtained results for s.p.d. band linear systems, and implicitily indicates the behaviour that can be expected from an execution with other problems.

6 Concluding Remarks

We have presented new hybrid CPU-GPU routines that accelerate the solution of s.p.d. band linear systems by offloading the computationally-expensive operations to the GPU. Our first CPU-GPU implementation computes the Cholesky factorization by executing the involved BLAS-3 kernels in the GPU, while maintaining a modest volume of CPU-GPU communication. The alternative CPU-GPU variant overlaps the Cholesky factorization with the solution of the first triangular band linear systems. Both variants rely on appropriate kernels from NVIDIA's CUBLAS and Intel's MKL.

The experimental results observed using several band test cases (with dimensions between 38,400 and 128,000 and a bandwidth of 1%, 2% and 4% of the problem size), in two platforms equipped with modern GPUs, reveal that our hybrid codes outperform the MKL routines for large matrices while they are still competitive for relatively small ones. Additionally, we have applied the new solvers to the solution of Lyapunov equations. The results show that the proposed Lyapunov solver outperforms its MKL counterpart in the solution of large problems, reaching a speed-up of 2×, when tackling a model order reduction problem of dimension 5,177.

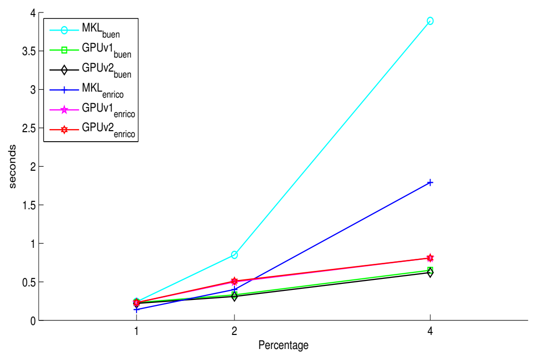
Acknowledgements Ernesto Dufrechou and Pablo Ezzatti acknowledge the support from *Programa de Desarrollo de las Ciencias Básicas*, and *Agencia Nacional de Investigación e Innovación*, Uruguay. Enrique S. Quintana-Ortí was supported by project TIN2011-23283 of the Ministry of Science and Competitiveness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

References

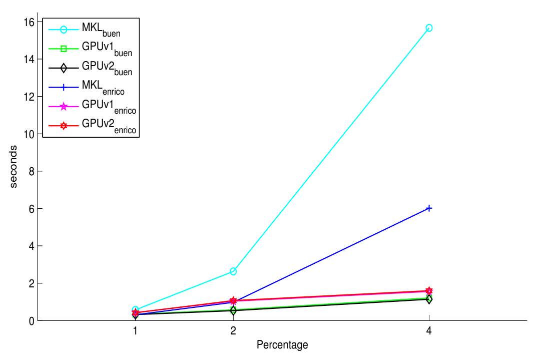
1. Anderson, E., Bai, Z., Demmel, J., Dongarra, J.E., DuCroz, J., Greenbaum, A., Hammarling, S., McKenney, A.E., Ostrouchov, S., Sorensen, D.: LAPACK Users' Guide. SIAM, Philadelphia (1992)
2. Antoulas, A.: Approximation of Large-Scale Dynamical Systems. SIAM Publications, Philadelphia, PA (2005)
3. Barrachina, S., Castillo, M., Igual, F.D., Mayo, R., Quintana-Ortí, E.S.: Solving dense linear systems on graphics processors. In: Euro-Par, pp. 739–748 (2008)
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**(18), 2457–2477 (2009)
5. Benner, P., Dufrechou, E., Ezzatti, P., Quintana-Ortí, E.S., Remón, A.: Accelerating Band Linear Algebra Operations on GPUs with Application in Model Reduction, *Lecture Notes in Computer Science*, vol. 8584. Springer (2014)
6. 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**(8), 1170–1182 (2013)
7. Benner, P., Kürschner, P., Saak, J.: Efficient handling of complex shift parameters in the low-rank Cholesky factor ADI method. Numerical Algorithms **62**(2), 225–251 (2013)
8. Benner, P., Kürschner, P., Saak, J.: Self-generating and efficient shift parameters in ADI methods for large Lyapunov and Sylvester equations. Electronic Transactions on Numerical Analysis **43**, 142–162 (2014)
9. Cuthill, E., McKee, J.: Reducing the bandwidth of sparse symmetric matrices. In: Proceedings of the 1969 24th

National Conference, ACM '69, pp. 157–172. ACM, New York, NY, USA (1969)

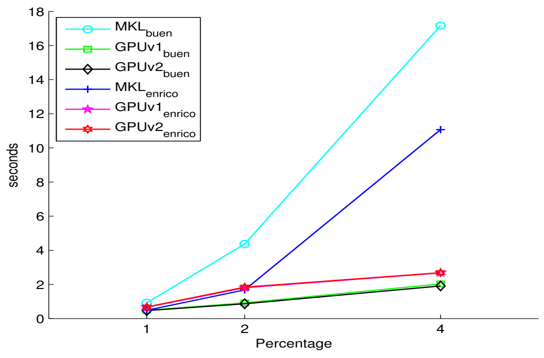
10. Du Croz, J., Mayes, P., Radicati, G.: Factorization of band matrices using level 3 BLAS. LAPACK Working Note 21, Technical Report CS-90-109, University of Tennessee (1990)
11. Dufrechou, E., Ezzatti, P., Quintana-Ortí, E., Remón, A.: Efficient symmetric band matrix-matrix multiplication on GPUs. Communications in Computer and Information Science **485**, 1–12 (2014)
12. Farber, R.: CUDA application design and development. Morgan Kaufmann (2011)
13. IMTEK: Oberwolfach model reduction benchmark collection. http://www.imtek.de/simulation/benchmark/
14. Khronos group: URL http://www.khronos.org/opencl
15. Kirk, D., Hwu, W.: Programming Massively Parallel Processors, Second Edition: A Hands-on Approach. Morgan Kaufmann (2012)
16. OpenACC.org: URL http://www.openacc-standard.org
17. Peise, E., Bientinesi, P.: Performance modeling for dense linear algebra. In: Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis, SCC '12, pp. 406–416. IEEE Computer Society, Washington, DC, USA (2012)
18. Penzl, T.: A cyclic low-rank Smith method for large sparse Lyapunov equations. SIAM J. Sci. Comput. **21**(4), 1401–1418 (1999)
19. The Top500 list: Available at http://www.top500.org (2013)
20. Volkov, V., Demmel, J.: LU, QR and Cholesky factorizations using vector capabilities of GPUs. Tech. Rep. UCB/EECS-2008-49, EECS Department, University of California, Berkeley (2008). URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-49.html



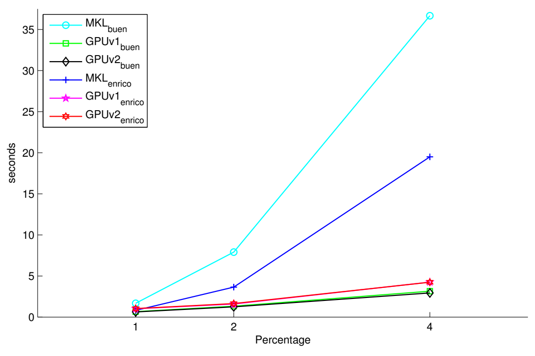
(a) $n = 38,400$



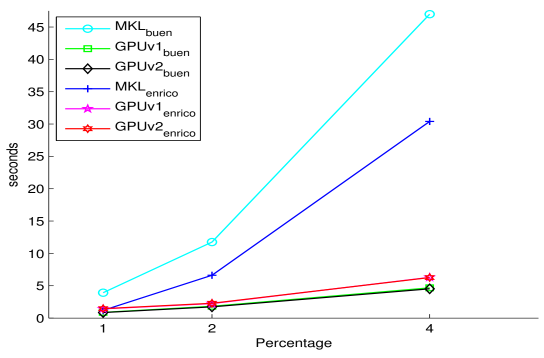
(b) $n = 51,200$



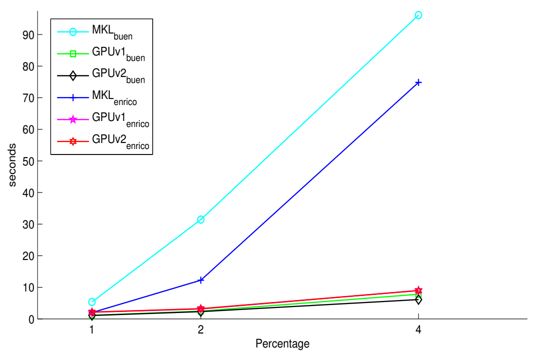
(c) $n = 64,000$



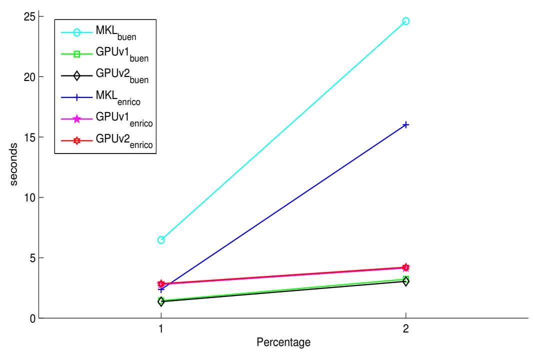
(d) $n = 76,800$



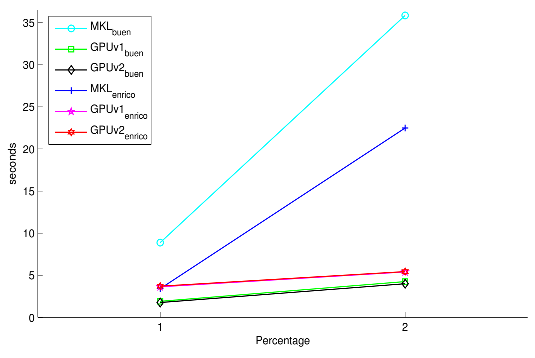
(e) $n = 89,600$



(f) $n = 102,400$



(g) $n = 115,200$



(h) $n = 128,000$

Fig. 7 Performance of the two versions of the hybrid CPU+GPU s.p.d. band system solver and Intel's MKL implementation.