CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **29**:e4055
Published online 3 March 2017 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.4055

SPECIAL ISSUE PAPER

Solving dense symmetric indefinite systems using GPUs

Marc Baboulin[1,*,†], Jack Dongarra[2], Adrien Rémy[1], Stanimire Tomov[2] and Ichitaro Yamazaki[2]

[1]*University of Paris-Sud, Orsay, France*
[2]*University of Tennessee, Knoxville, USA*

SUMMARY

This paper studies the performance of different algorithms for solving a dense symmetric indefinite linear system of equations on multicore CPUs with a Graphics Processing Unit (GPU). To ensure the numerical stability of the factorization, *pivoting* is required. Obtaining high performance of such algorithms on the GPU is difficult because all the existing pivoting strategies lead to frequent synchronizations and irregular data accesses. Until recently, there has not been any implementation of these algorithms on a hybrid CPU/GPU architecture. To improve their performance on the hybrid architecture, we explore different techniques to reduce the expensive data transfer and synchronization between the CPU and GPU, or on the GPU (e.g., factorizing the matrix entirely on the GPU or in a communication-avoiding fashion). We also study the performance of the solver using iterative refinements along with the factorization without pivoting combined with the preprocessing technique based on random butterfly transformations, or with the mixed-precision algorithm where the matrix is factorized in single precision. This randomization algorithm only has a probabilistic proof on the numerical stability, and for this paper, we only focused on the mixed-precision algorithm without pivoting. However, they demonstrate that we can obtain good performance on the GPU by avoiding the pivoting and using the lower precision arithmetics, respectively. As illustrated with the application in acoustics studied in this paper, in many practical cases, the matrices can be factorized without pivoting. Because the componentwise backward error computed in the iterative refinement signals when the algorithm failed to obtain the desired accuracy, the user can use these potentially unstable but efficient algorithms in most of the cases and fall back to a more stable algorithm with pivoting only in the case of the failure. Copyright © 2017 John Wiley & Sons, Ltd.

Received 15 March 2016; Revised 28 September 2016; Accepted 18 October 2016

KEY WORDS: dense symmetric indefinite systems; symmetric pivoting; communication-avoiding; randomization; GPU computation; iterative refinement

1. INTRODUCTION

A symmetric matrix A is called indefinite when the quadratic form $x^T Ax$ can take both positive and negative values. Dense linear systems of equations with symmetric indefinite matrices appear in many studies of physics, including physics of structures, acoustics, and electromagnetism. For instance, such systems arise in the linear least-squares problem for solving an augmented system [1, p. 77], or in the electromagnetism where the discretization by the boundary element method results in linear systems with dense complex symmetric (non-Hermitian) matrices [2]. The efficient solution of these linear systems demands a high-performance implementation of a dense symmetric indefinite solver that can efficiently use the current hardware architecture. In particular, the use of accelerators has become pervasive in scientific computing because of their high-performance

*Correspondence to: Marc Baboulin, University of Paris-Sud, Orsay, France.
†E-mail: baboulin@lri.fr

capabilities and low-energy consumptions. For example, in terms of the floating-point operation per second, or flop/s in short, a single K40 NVIDIA GPU has a double precision peak performance of 1689 Gflop/s for a thermal design power of 235 W. According to benchmarks in the MAGMA library [3], optimized large-dense matrix computations, for example, matrix–matrix multiplications, reach 1200 Gflop/s for a power draw of about 200 W, that is, $\approx$ 6 Gflop/W. In contrast, two Sandy Bridge E5-2670 CPUs have about the same thermal design power ($2 \times 115 = 230$ W) as the K40 but for a peak of 333 Gflop/s, which translates to only 1.4 Gflop/W for the Sandy Bridge CPU. To achieve the high performance, however, the algorithms must be designed for high parallelism and high "flops to data" ratio while maintaining a low number of flops and exploiting the hardware features of the hybrid CPU/GPU architecture. A dense symmetric indefinite solver that can efficiently exploit the GPU's high-computing power would be useful for many physical applications.

To solve a symmetric indefinite linear system of equations, $Ax = b$, a classical method decomposes the matrix A into an LDL^T factorization,

$$PAP^T = LDL^T, \tag{1}$$

where L is unit lower triangular, D is block diagonal with either 1-by-1 or 2-by-2 diagonal blocks, and P is a permutation matrix to ensure the numerical stability of the factorization. Then the solution x is computed by successively solving the linear systems with the coefficient matrices L, D, and L^T along with the permutation. The strategies to compute the permutation matrix P for the LDL^T factorization include complete pivoting (Bunch–Parlett algorithm) [4], partial pivoting (Bunch–Kaufman algorithm) [5], rook pivoting (bounded Bunch–Kaufman) [6, p. 523], and fast Bunch–Parlett [6, p. 525]. In particular, the Bunch–Kaufman and rook pivoting strategies are implemented in LAPACK [7], a set of dense linear algebra routines on multicore CPUs, that are extensively used in many scientific and engineering simulations. The routines implemented in LAPACK are based on block algorithms that can exploit the memory hierarchy on modern architectures, using BLAS-3 matrix operations for most of its floating-point operations.

Another promising method for solving a symmetric indefinite linear system is the Aasen's method [8], which computes the LTL^T factorization of the matrix A,

$$PAP^T = LTL^T, \tag{2}$$

where T is now a symmetric tridiagonal matrix. The algorithm requires $\frac{1}{3}n^3 + O(n^2)$ flops [9, p. 166], similarly to the LDL^T factorization. A block algorithm for computing the LTL^T factorization was also proposed [10]. Although the block implementation performs slightly more flops (i.e., an additional rank-1 update of the trailing submatrix, Section 2.3), it can exploit a modern computer's memory hierarchy and obtain performance similar to the Bunch–Kaufman algorithm implemented in LAPACK [10].

To maintain numerical stability, the pivoting techniques mentioned earlier involve between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons to search for pivots and possible interchanges of selected columns and rows. Hence, factorizing each column of the matrix requires the synchronization for selecting the pivot and the data movement for exchanging the columns and rows, which have become significantly more expensive compared with the arithmetic operations on modern computers. Furthermore, because only either the upper or lower triangular part of the matrix A is stored, the symmetric pivoting[‡] requires irregular data access (i.e., some parts of the pivot column may be stored as the transpose of the corresponding part of the row), which dramatically increases the cost of the data movement. Partially because of these performance challenges, ScaLAPACK [11], which is the extension of LAPACK for distributed-memory machines, does not support the symmetric indefinite factorization, and until recently, there were no implementations of the algorithm that could exploit a GPU[§]. This motivated our efforts to review the different factorization algorithms, develop their efficient implementations on multicores with a GPU to address their current limitations, and show the

[‡]To maintain the symmetry, both columns and rows must be swapped.

[§]A Bunch–Kaufman implementation became recently available in the cuSolver library as part of the CUDA Toolkit v7.5 from NVIDIA.

new state-of-the-art outlook for this important problem. For example, recently, a communication-avoiding variant of the Aasen's algorithm was proposed [12]. However, the pivoting must still be applied symmetrically, leading to expensive irregular data accesses. Another technique studied in this paper is a symmetric version of random butterfly transformations (RBT) [13] on the GPU. RBT can be combined with an LDL^T factorization to probabilistically improve the stability of the factorization without pivoting. The performance of RBT for symmetric indefinite systems has been studied on multicore systems [14] and distributed-memory systems [15], but its performance has not been investigated on a GPU. Finally, we study the potential of a mixed-precision algorithm to improve the performance of the solver, where the matrix is first factorized in single precision, and the solution is computed through iterative refinement.

This paper is organized as follows. Section 2 describes the three algorithms for solving dense symmetric indefinite systems (i.e., the Bunch–Kaufman and Aasen's algorithms, and the RBTs) and their implementations on the hybrid CPU/GPU architecture. It also explains how we can use mixed precision to accelerate the solver. Section 3 shows our experimental results, where Sections 3.1 and 3.2 present the performance and numerical results for random matrices and two acoustic scattering problems, respectively, while Section 3.3 gives performance results of the mixed-precision algorithm applied to random matrices without pivoting. Section 4 contains concluding remarks. In this paper, we use $a_{i,j}$ and a_j to denote the (i,j)-th entry and the j-th column of the matrix A, respectively, while $A_{i_1:i_2,j_1:j_2}$ is the submatrix consisting of the i_1-th through the i_2-th rows and the j_1-th through the j_2-th columns of A. We also use $A_{I,J}$ and $A_{I_1:I_2,J_1:J_2}$ to denote the (I,J)-th block and the submatrix consisting of the I_1-th through the I_2-th block rows and J_1-th through the J_2-th block columns of A, where the block size is n_b and the number of block columns/rows in A is n_t (i.e., $n_t = \lceil \frac{n}{n_b} \rceil$).

This paper extends our previous proceedings paper [16] presented at the PPAM 2015 conference. In this extended paper, we describe the current general trends in designing efficient numerical linear algebra libraries on manycore accelerated architectures (Section 2.1) before presenting our specific design and optimization of the symmetric indefinite solvers for the GPU architectures (Section 2.2). We also include the time to obtain the solution while the previous paper only showed the factorization time and give more details about the acoustic scattering problems studied in the paper (Sections 3.1 and 3.2, respectively). Finally, we describe our implementation and its performance of the mixed-precision algorithm that may improve the performance of the solver in practice (Sections 2.5 and 3.3).

2. SYMMETRIC INDEFINITE FACTORIZATIONS WITH A GPU

In this section, we describe the existing algorithms for solving a dense symmetric indefinite linear system of equations. First, we describe the general principles for designing an efficient dense linear algebra algorithm on heterogeneous systems, and then, we concentrate on the specifics for the design and optimization of symmetric indefinite solvers for GPU architectures, along with discussion on what design principles can (or cannot) be applied for these solvers.

2.1. Programming linear algebra solvers on GPUs

The LAPACK's programming model [7] is based on expressing algorithms in terms of BLAS calls. Subsequently, LAPACK can achieve high efficiency, provided that highly efficient BLAS implementations are provided on the target machine, for example, by the manufacturer. Since the 1980s, this model has turned out to be very successful for cache-based shared-memory vector and parallel processors with multi-layered memory hierarchies.

To account for the deep memory hierarchies today, efficient BLAS implementations feature multilevel blocking where, for example, the Level 3 matrix–matrix computations are split hierarchically into blocks that fit into corresponding levels of the memory hierarchy [17]. In effect, a programming model based on BLAS is still an effective model for exploiting the deep memory hierarchies at the present time [18]. However, the resulting parallelism is *fork-join* – a sequence of BLAS calls is implicitly synchronized after each individual BLAS call (join), although the routines by themselves run in parallel (fork). This brings synchronization overheads and idle time for some processors/cores, especially on the highly parallel current and future heterogeneous system designs [19], motivating the search for improved models where the BLAS routines are broken into small tasks and properly scheduled for execution over the heterogeneous hardware components.

The typical hybrid algorithm splits the overall computation into small tasks to execute on the CPU, and large update tasks to execute on the accelerator [3, 20–22]. For instance, in LU and QR factorizations, each step is split into a panel factorization of n_b columns, followed by a trailing matrix update. The panel factorization is assigned to the CPU and includes such decisions as selecting the maximum pivot in each column or computing a Householder reflector for each column. The trailing matrix update is assigned to the accelerator and involves some form of matrix–matrix multiply. The block size, n_b, can be tuned to adjust the amount of work on the CPU versus on the accelerator. Optimally, during the trailing matrix update, a look-ahead panel is updated first and sent back to the CPU. Asynchronous data transfers are used to copy data between the CPU and accelerator, while the accelerator continues computing. The CPU performs the next panel factorization, while the accelerator continues with the remainder of the trailing matrix update. In this way, the inputs for the next trailing matrix update are ready when the current update finishes. The goal is to keep the accelerator always busy, which has the highest performance.

Unfortunately, the pivoting required to maintain the numerical stability of the symmetric indefinite factorization leads to the fork-join and prohibits the look-ahead as we describe in the rest of this section.

2.2. Bunch–Kaufman algorithm

One of the most widely used algorithms for solving a symmetric indefinite linear system is based on the block LDL^T factorization with the Bunch–Kaufman algorithm [5], which is also implemented in LAPACK (i.e., `xSYTRF`). The pseudo-code of the algorithm is shown in Figure 1(a), which is referred to as a *right-looking* algorithm because after the set of n_b columns, commonly referred to as *panel*, are factorized, the panel is used to update the trailing submatrix, which is on the right of the panel. To select the pivot at each step of the factorization, it scans at most two columns of the trailing submatrix, and depending on the numerical values of the scanned matrix entries, it uses either a 1-by-1 or a 2-by-2 pivot. This algorithm has satisfactory backward stability [23, p. 219]. Then a variant of the Bunch–Kaufman algorithm, also called "rook pivoting," was proposed in [6] that provides a better accuracy by bounding the triangular factors. However, depending on the matrix, the rook pivoting method could perform $O(n^3)$ comparisons, as opposed to the $O(n^2)$ comparisons of the Bunch–Kaufman algorithm. Hence, in this paper, we focus on the Bunch–Kaufman algorithm as a baseline for our performance comparison.

Our implementation of the Bunch–Kaufman algorithm on the hybrid architecture is based on BLAS and LAPACK task representations (as described in Section 2.1), where the BLAS and LAPACK calls on the CPU are replaced with the corresponding GPU kernels (Figure 2). In addition, our first implementation is based on a hybrid CPU/GPU programming paradigm where the panel is factorized on the CPU (e.g., using the multithreaded MKL library [24]), while the trailing submatrix is updated on the GPU. This is often an effective programming paradigm for many of the LAPACK subroutines because the panel factorization is based on BLAS-1 or BLAS-2, which can be efficiently implemented on the CPU, while BLAS-3 is used for the submatrix updates, which exhibit high-data parallelism and can be efficiently implemented on the GPU [3, 25]. Unfortunately, at each step of the panel factorization, the Bunch–Kaufman algorithm may select the pivot from the trailing submatrix. Hence, although copying the panel from the GPU to the CPU can be overlapped with the update of the rest of the trailing submatrix on the GPU, the *look-ahead* – a standard optimization technique to overlap the panel factorization on the CPU with the trailing submatrix update on the GPU – is prohibited. In addition, when the pivot column is on the GPU, this leads to an expensive data transfer between the GPU and the CPU at each step of the factorization. To avoid this expensive data transfer, our second implementation performs the entire factorization on the GPU. Although the

```
$\alpha = (1+\sqrt{17})/8$ and $j = 1$
while $j < n$ do
  $k = j$
  {Panel factorization}
  while $j < k + n_b - 1$ do
    Update column $a_j$ with previous columns
    $r = \arg\max_{i>j} |a_{i,j}|$ and $\gamma = |a_{r,j}|$
    if $\gamma > 0$ then
      if $|a_{j,j}| \geq \alpha\gamma$ then
        $s = 1$
        Use $a_{j,j}$ as a $1 \times 1$ pivot.
      else
        Update $a_r$ with previous columns
        $\omega = \max_{i \geq j; i \neq r} |a_{i,r}|$
        if $|a_{j,j}|\omega \geq \alpha\gamma^2$ then
          $s = 1$
          Use $a_{j,j}$ as a $1 \times 1$ pivot.
        else
          if $|a_{r,r}| \geq \alpha\omega$ then
            $s = 1$
            Swap rows/columns $(j, r)$
            Use $a_{r,r}$ as a $1 \times 1$ pivot.
          else
            $s = 2$
            Swap rows/columns $(j+1, r)$
            Use $\begin{pmatrix} a_{j,j} & a_{r,j} \\ a_{r,j} & a_{rr} \end{pmatrix}$ as $2 \times 2$
            pivot.
          end if
        end if
      end if
    else
      $s = 1$
    end if
    Scale the pivot columns to compute $L_{j:j+s-1}$
    $j = j + s$
  end while
  {Right-looking trailing submatrix update}
  $A_{j:n,j:n} := A_{j:n,j:n} - L_{j:n,k:j} D_{k:j,k:j} L^T_{j:n,k:j}$
end while
```

(a) Bunch-Kaufman [14].

```
for $J = 1, 2, \ldots, n_t$ do
  for $I = 2, 3, \ldots, J-1$ do
    $X = T_{I,I-1} L^T_{J,I-1}$
    $Y = T_{I,I} L^T_{J,I}$
    $Z = T_{I,I+1} L^T_{J,I+1}$
    $W_{I,J} = 0.5Y + Z$
    $H_{I,J} = X + Y + Z$
  end for

  $A_{J,J} = A_{J,J} - L_{J,2:J-1} W_{2:J-1,J}$
             $- W^T_{2:J-1,J} L^T_{J,2:J-1}$
  $T_{J,J} = L^{-1}_{J,J} A_{J,J} L^{-T}_{J,J}$
  if $J < n_t$ then
    if $J > 1$ then
      $H_{J,J} = T_{J,J-1} L^T_{J,J-1} + T_{J,J} L^T_{J,J}$
    end if

    $E = A_{J+1:n_t,J} - L_{J+1:n_t,2:J} H_{2:J,J}$
    $[L_{J+1:n_t,J+1}, H_{J+1,J}, P^{(J)}] = \mathrm{LU}(E)$

    $T_{J+1,J} = H_{J+1,J} L^{-T}_{J,J}$

    $L_{J+1:n_t,2:J} = P^{(J)} L_{J+1:n_t,2:J}$
    $A_{J+1:n_t,J+1:n_t} = P^{(J)} A_{J+1:n_t,J+1:n_t} P^{(J)T}$
    $P_{J+1:n_t,1:n_t} = P^{(J)} P_{J+1:n_t,1:n_t}$
  end if
end for
```

(b) CA Aasen's [11], where the first block column $L_{1:n_t,1}$ is the first n_b columns of the identity matrix and $[L, U, P] = \mathrm{LU}(A)$ returns the LU factors of A with partial pivoting such that $LU = PA$.

Figure 1. Symmetric indefinite factorization algorithm: (a) Bunch–Kaufman [5]; (b)A Aasen's [12], where the first block column $L_{1:n_t,1}$ is the first n_b columns of the identity matrix and $[L, U, P] = \mathrm{LU}(A)$ returns the LU factors of A with partial pivoting such that $LU = PA$.

CPU may be more efficient at performing the BLAS-1 and BLAS-2 based panel factorization, this implementation often obtains higher performance by avoiding the expensive data transfer (Figure 4).

When the entire factorization is implemented on the GPU, up to two columns of the trailing submatrix must be scanned to select a pivot – the current column and the column with index corresponding to the row index of the element with the maximum modulus in the first column. This not only leads to the expensive global reduce on the GPU but also to irregular data accesses because only the lower-triangular part of the submatrix is stored. This makes it difficult to obtain high performance on the GPU. In the next two sections, we describe two other algorithms (i.e., communication-avoiding and randomization algorithms) that aim at reducing this bottleneck.

2.3. *Aasen's algorithm*

To solve a symmetric indefinite linear system, Aasen's algorithm [8] factorizes A into an LTL^T decomposition. The algorithm takes advantage of the symmetry of A and performs $\frac{1}{3}n^3 + O(n^2)$ flops, which are the same flop count as that of the Bunch–Kaufman algorithm. In addition, like the Bunch–Kaufman algorithm, it is backward stable subject to a growth factor. To maintain the stability, at each step of the factorization, it uses the largest element of the current column being factorized

```
for (int k = 0; k < min(nb-1,n); k += kstep) {
  /* Copy column K of A to column K of W and update it */
  magmablasSetKernelStream( queues[0] );
  magma_dcopy( n-k, &dA( k, k ), 1, &dW( k, k ), 1 );
  magma_dgemv( MagmaNoTrans, n-k, k, c_mone, &dA( k, 0 ), ldda,
          &dW( k, 0 ), lddw, c_one, &dW( k, k ), ione );
  kstep = 1;

  /* Determine rows and columns to be interchanged */
  magma_dgetvector_async( 1, &dW( k, k ), 1, &Z, 1, queues[0] );
  abs_akk = fabs( Z );
  if ( k < n-1 ) {
    imax = k + magma_idamax( n-k-1, &dW(k+1,k), 1 );
    magma_dgetvector( 1, &dW( imax, k ), 1, &Z, 1 );
    colmax = MAGMA_D_ABS1( Z );
  } else {
    colmax = d_zero;
  }

  if ( max( abs_akk, colmax ) == 0.0 ) {
    /* Column K is zero: set INFO and continue */
    if ( *info == 0 ) *info = k;
    kp = k;
  } else {
    if ( abs_akk >= alpha*colmax ) {
      /* no interchange, use 1-by-1 pivot block */
      kp = k;
    } else {
      /* Copy column imax to column K+1 of W and update it */
      magma_dcopy( imax-k, &dA( imax, k ), ldda, &dW( k, k+1 ), 1 );
      magma_dcopy( n-imax, &dA( imax, imax ), 1, &dW( imax, k+1 ), 1 );
      magma_dgemv( MagmaNoTrans, n-k, k, c_mone, &dA( k, 0 ), ldda,
              &dW( imax, 0 ), lddw, c_one, &dW( k, k+1 ), ione );
      magma_dgetvector_async( 1, &dW( imax, k+1 ), 1,
              &Zimax, 1, queues[0] );

      jmax = k-1 + magma_idamax( imax-k, &dW(k, k+1), 1 );
      magma_dgetvector( 1, &dW( jmax, k+1 ), 1, &Z, 1 );
      rowmax = MAGMA_D_ABS1( Z );
      if ( imax < n-1 ) {
        jmax = imax + magma_idamax( (n-1)-imax, &dW( imax+1, k+1 ), 1 );
        magma_dgetvector( 1, &dW( jmax, k+1 ), 1, &Z, 1 );
        rowmax = max( rowmax, MAGMA_D_ABS1( Z ) );
      }

      if ( abs_akk >= alpha*colmax*( colmax / rowmax ) ) {
        /* no interchange, use 1-by-1 pivot block */
        kp = k;
      } else if ( fabs( Zimax ) >= alpha*rowmax ) {
        /* interchange rows and columns K and imax, use 1-by-1
        pivot block */
        kp = imax;
        /* copy column K+1 of W to column K */
        magma_dcopy( n-k, &dW( k, k+1 ), 1, &dW( k, k ), 1 );
      } else {
        /* interchange rows and columns K+1 and imax, use 2-by-2
        pivot block */
        kp = imax;
        kstep = 2;
      }
    }
    kk = k + kstep - 1;
    /* Updated column kp is already stored in column kk of W */
    if ( kp ! = kk ) {
      /* Copy non-updated column kk to column kp */
      magma_dcopy( kp-kk, &dA( kk, kk ), 1, &dA( kp, kk ), ldda );
      magma_dcopy( n-kp, &dA( kp, kk ), 1, &dA( kp, kp ), 1 );

      /* Interchange rows kk and kp in first kk columns of A and W */
      magmablas_dswap( kk+1, &dA( kk, 0 ), ldda, &dA( kp, 0 ), ldda );
      magmablas_dswap( kk+1, &dW( kk, 0 ), lddw, &dW( kp, 0 ), lddw );
    }
```

(a) Bunch-Kaufman.

```
      if ( kstep == 1 ) {
        /* 1-by-1 pivot */
        magma_dcopy( n-k, &dW( k, k ), 1, &dA( k, k ), 1 );

        if ( k < n-1 ) {
          magma_dgetvector_async( 1, &dA( k, k ), 1, &Z, 1, queues[0] );
          R1 = d_one / Z;
          magma_dscal((n-1)-k, R1, &dA( k+1, k ), 1);
        }
      } else {
        /* 2-by-2 pivot */
        if (n > k+2)
        magmablas_dlascl_2x2( MagmaLower, n-(k+2),
                &dW(k,k), lddw, &dA(k+2,k), ldda, &iinfo );

        /* Copy D(k) to A */
        magma_dcopymatrix( 2,2, &dW( k, k ), lddw, &dA( k, k ), ldda );
      }
    }
    /* Store details of the interchanges in ipiv */
    if ( kstep == 1 ) {
      ipiv[k] = kp+1;
    } else {
      ipiv[k] = -kp-1;
      ipiv[k+1] = -kp-1;
    }
  }
/* Update the lower triangle of trailing submatrix */
magma_event_record( events[0], queues[0] );
for (int queue_id = 1; queue_id<num_queues-1; queue_id ++) {
  magma_queue_wait_event( queues[queue_id], events[0] );
}
for( int j = k; j < n; j += nb ) {
  int jb = min( nb, n-j );
  int queue_id = (j-k)%(num_queues-1);
  magmablasSetKernelStream( queues[queue_id] );
  magma_dgemm( MagmaNoTrans, MagmaTrans, n-j, jb, k,
               c_mone,  &dA( j, 0 ), ldda,
                        &dW( j, 0 ), lddw,
               c_one,   &dA( j, j ), ldda );
}

/* Put L21 in standard form by undoing row interchanges */
for (int queue_id=1; queue_id<num_queues-1; queue_id++) {
  magma_event_record( events[queue_id], queues[queue_id] );
  magma_queue_wait_event( queues[0], events[queue_id] );
}
for (int j = k; j > 0; ) {
  int jj = j;
  int jp = ipiv[j-1];
  if ( jp < 0 ) {
    jp = -jp;
    j -= 1;
  }
  j -= 1;
  if ( jp ! = jj && j >= 1 ) {
    magmablasSetKernelStream( queues[0] );
    magmablas_dswap( j, &dA( jp-1, 0 ), ldda, &dA( jj-1, 0 ), ldda );
    magma_queue_sync( queues[0] );
  }
}
// copying the panel back to CPU
magma_event_record( events[0], queues[0] );
magma_queue_wait_event( queues[num_queues-1], events[0] );
magma_dgetmatrix_async( n,k,
          &dA(0,0),ldda, &A(0,0),lda, queues[num_queues-1] );
```

(b) Bunch-Kaufman (continued).

Figure 2. Graphics processing unit implementation of Bunch–Kaufman algorithm: (a) Bunch–Kaufman; (b) Bunch–Kaufman (continued).

as the pivot, leading to more regular data access compared with the Bunch–Kaufman algorithm (that may scan an additional column, some part of which may be stored as the transpose of the corresponding part of the row). To exploit the memory hierarchy of modern computers, a blocked version of the algorithm was developed [10], which is based on a *left-looking* panel factorization, followed by a right-looking trailing submatrix update using BLAS-3 routines. Compared with the column-wise algorithm, this blocked algorithm performs slightly more flops, requiring $\frac{1}{3}(1+\frac{1}{n_b})n^3 + O(n^2 n_b)$ flops with a block size n_b, but BLAS-3 can be used to perform most of these flops (i.e., $\frac{1}{3}(1+\frac{1}{n_b})n^3$). However, the panel factorization is still based on BLAS-1 and BLAS-2, which often obtains only a small fraction of the peak performance. To improve the performance of the panel factorization, another variant of the algorithm was proposed [12]. This other variant computes an LTL^T factorization of A, where T is a banded matrix with its half-bandwidth equal to the block size n_b and then uses a banded matrix solver to compute the solution. This algorithm factorizes each panel using an existing LU factorization algorithm, such as recursive LU [26–28] or communication-avoiding LU (TSLU, for the panel) [29, 30]. In comparison with the panel factorization algorithm used in the block Aasen's algorithm, these LU factorization algorithms reduce communication and are likely to speed up the whole factorization process. This is referred to as a communication-avoiding (CA) variant of the Aasen's algorithm, and its pseudocode is shown in Figure 1(b).

In general, a GPU has a greater memory bandwidth than a CPU, but the memory accesses are still expensive compared with the arithmetic operations. Hence, our implementation is based on the CA Aasen's algorithm. Although this algorithm performs most of the flops using BLAS-3 (e.g., xGEMM), most of the operations are on the submatrices of the dimension n_b-by-n_b. In order to run these small independent BLAS calls in parallel on the GPU, we use GPU streams. An alternative is to use Batched BLAS, where all independent xGEMMs are grouped together in a single call. Implementations are available in both MAGMA and CUBLAS. However, as shown in [31, Figure 8(f)], the streamed implementation (that we use here) is faster than either the MAGMA Batched or CUBLAS Batched DGEMM for matrices of size above 160 (on K40c GPU for DGEMM on square matrices, i.e., $m = n = k$), which is the case here. With the GPU streams, the CA Aasen obtained its best performance using $n_b = 256$ (Figure 4).

Our CA Aasen's implementation applies the pivots in two steps: The first step copies all the columns of the trailing submatrix, which needs to be swapped, into an n-by-$2n_b$ workspace. Here, because of the symmetry, the k-th block column consists of the blocks in the k-th block row and those in the k-th block column (each block column consists of the n_b contiguous columns). Then, in the second step, we copy the columns of the workspace back to a block column of the submatrix after the column pivoting is applied. The two-step implementation is used to exploit the parallelism on multicore CPU [32] and in our non-GPU-resident implementations to factorize the matrices that do not fit in the GPU memory at once [33]. In our experiments, to factorize the panel, we used the LU factorization with partial pivoting, using either the multithreaded MKL library on the CPU or using its native GPU implementation in MAGMA on the GPU. Although the BLAS-1 and BLAS-2 based panel factorization may be more efficient on the CPU, the second approach avoids the expensive data transfer required to copy the panel from the GPU to the CPU (see Section 3 for the performance results).

2.4. *Random butterfly transformations*

Random butterfly transformation is a randomization technique initially described by Parker [13] and recently revisited for dense linear systems, either general [34] or symmetric indefinite [15]. It has also been applied recently to a sparse direct solver in [35]. The procedure to solve $Ax = b$, where A is a symmetric indefinite matrix, using a random transformation and the LDL^T factorization is summarized in Algorithm 1. The random matrix U is chosen among a particular class of matrices called *recursive butterfly matrices*. A *butterfly matrix* is an $n \times n$ matrix of the form

$$B^{<n>} = \frac{1}{\sqrt{2}} \begin{bmatrix} R_0 & R_1 \\ R_0 & -R_1 \end{bmatrix}$$

where R_0 and R_1 are random diagonal $\frac{n}{2} \times \frac{n}{2}$ matrices. A *recursive butterfly matrix* of size n and depth d is defined recursively as

$$W^{<n,d>} = \begin{bmatrix} B_1^{<n/2^{d-1}>} & & \\ & \ddots & \\ & & B_{2^{d-1}}^{<n/2^{d-1}>} \end{bmatrix} \cdot W^{<n,d-1>}, \text{ with } W^{<n,1>} = B^{<n>}$$

where the $B_i^{<n/2^{d-1}>}$ are butterflies of size $n/2^{d-1}$, and $B^{<n>}$ is a butterfly of size n. The application of RBT to symmetric indefinite problems was studied in [36] where it is shown that in practice,

$d = 1$ or 2 gives satisfactory results. Note that, as mentioned in [34], the solution can be improved by adding systematically some steps of iterative refinement in the working precision as indicated in [23, p. 232]. It is also shown that random butterfly matrices are cheap to store and apply ($O(nd)$ and $O(dn^2)$, respectively). An implementation for the multicore library PLASMA was described in [14].

Algorithm 1 Random butterfly transformation algorithm

```
Generate recursive butterfly matrix $U$
Apply randomization to update the matrix $A$ and compute the matrix $A_r = U^T AU$
Factorize the randomized matrix using $LDL^T$ factorization with no pivoting
Compute right-hand side $U^T b$, solve $A_r y = U^T b$, then $x = Uy$
```

For the GPU implementation, we use a recursive butterfly matrix U of depth $d = 2$. Only the diagonal values of the blocks are stored into a vector of size $2 \times N$ as described in [34]. Applying the depth 2 recursive butterfly matrix U consists of multiple applications of depth 1 butterfly matrices on different parts of the matrix A. The application of a depth 1 butterfly matrix is performed using a CUDA kernel where the computed part of the matrix A is split into blocks. For each of these blocks, the corresponding part of the matrix U is stored in the shared memory to improve the memory access performance. Matrix U is small enough to fit into the shared memory due to its packed storage.

To compute the LDL^T factorization of A_r without pivoting, we implemented a block factorization algorithm on multicore CPUs with a GPU. In our implementation, the matrix is first copied to the GPU; then, the CPU is used to compute the LDL^T factorization of the diagonal block. Once the resulting LDL^T factors of the diagonal block are copied back to the GPU, the corresponding off-diagonal blocks of the L-factor are computed by the triangular solve on the GPU. Finally, we update each block column of the trailing submatrix calling a matrix–matrix multiply on the GPU.

2.5. *Mixed precision algorithm*

On modern computers, single precision 32-bit floating point arithmetic is usually at least twice as fast as double precision 64-bit floating point arithmetic. For example, on a latest NVIDIA GPU (e.g., the GeForce GTX Titan Black), the single precision peak performance is about 3× greater than the double precision peak peformance. This gap can be much greater depending on the number of 32-bit and 64-bit CUDA cores (e.g., 32× faster on the Titan X). To take advantage of this hardware trend for solving a linear system of equations, the mixed-precision algorithm may compute a solution in single precision and then aims to refine the solution to have double precision accuracy by performing only the critical parts of the algorithm in double precision. Iterative refinement in single/double precision is presented in [37–39] and has been implemented in so-called mixed precision solvers in [40, 41].

```
compute $LDL^T := A$          in single precision
solve $Ly = b$ for $y$            in single precision
solve $Dz = y$ for $z$            in single precision
solve $L^T x = z$ for $x$         in single precision
compute $r := b - Ax$         in double precision
while $\|r\|_2 > \|x\|_2 \|A\|_\infty \epsilon \sqrt{n}$ do
    solve $Ly = r$ for $y$        in single precision
    solve $Dz = y$ for $z$        in single precision
    solve $L^T e = z$ for $e$     in single precision
    compute $x := x + e$      in double precision
    compute $r := b - Ax$     in double precision
end while
```

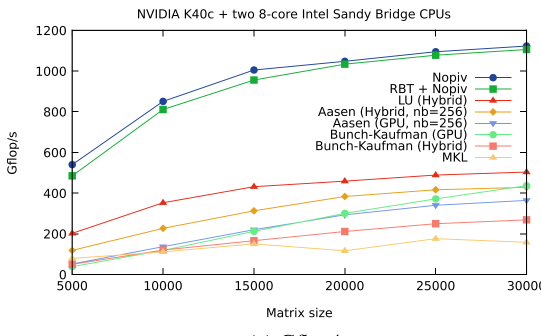
Figure 3. Fixed precision iterative refinement without pivoting where ϵ is the relative machine precision in double precision, given by LAPACK's DLAMCH. The algorithm can be trivially extended to use pivoting.

Figure 3 shows the pseudocode of such a mixed-precision algorithm, applied to the LDL^T factorization with no pivoting. Note that this is different from what is called “mixed precision” in the literature (e.g., [9, p. 127]) because in our case $x := x + e$ is computed in double precision. The factorization of the coefficient matrix A is the most computationally expensive kernel, requiring $\mathcal{O}(n^3)$ flops, while the other kernels require at most $\mathcal{O}(n^2)$ flops. To take advantage of the higher performance, the coefficient matrix A is converted to single precision and factorized in single precision. Then, in order to obtain double-precision accuracy, double-precision arithmetic is used to compute the residual vector and to update the solution vector. To compute the residual vector, the original coefficient matrix A is needed. Hence, compared with the standard algorithm, which performs all the operations in double precision, the mixed-precision algorithm requires 50% more memory to store A in single precision. However, the most expensive kernel is handled in single precision, and the mixed-precision algorithm may obtain a higher performance than the standard algorithm does, as long as it requires a small number of iterations. The numerical analysis of the standard or mixed-precision iterative refinements can be found in [9, 37–39, 42].

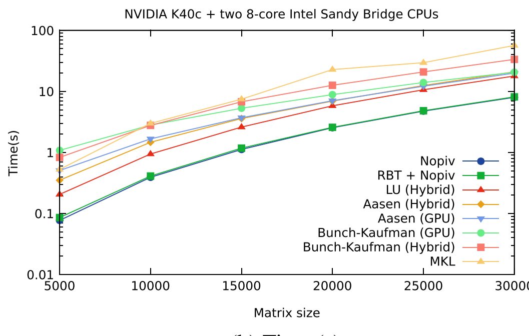
3. EXPERIMENTAL RESULTS

3.1. Comparison of symmetric indefinite solvers

Figure 4(a) and (b) compares respectively the performance in Gflop/s and time for the symmetric indefinite factorizations where the test matrices are random. The “Gflop/s” is computed as the ratio of the number of flops required for the LDL^T factorization (i.e., $n^3/3$) over time (in seconds) for the particular dimension of the matrix, n. Note that, for normalization of the graph, we also consider the same flop count for LU, even though it performs twice more flops. The experiments were conducted on two eight-core Intel SandyBridge CPUs with an NVIDIA K40c GPU. The code is compiled using the GNU `gcc` version 4.4.7 and the `nvcc` version 7.0 with the optimization flag `-O3` and linked with Intel’s Math Kernel Library (MKL) version xe_2013_sp1.2.144. First, when the matrix size is large enough (i.e., $n > 10{,}000$), the performance of the Bunch–Kaufman algorithm can be improved using the GPU over the multithreaded MKL implementation (routine `dsytrf`) on the 16 cores of two Sandy Bridge CPUs. In addition, performing the panel factorization on the GPU avoids the expensive data transfer between the CPU and GPU and may improve the performance of the hybrid CPU/GPU implementation. Next, the communication-avoiding variant of the Aasen’s algorithm further improves the performance of the Bunch–Kaufman by reducing the synchronization and communication costs required for selecting the pivots. The RBT approach outperforms the Bunch–Kaufman and Aasen factorizations, but, as mentioned infig:perffacto [14], it may not be numerically stable for some matrices, and it requires in general a few steps of iterative refinement in the working precision. However, the performance of all the symmetric factorizations with provable stability was lower than that of the LU factorization. In addition, although our current implementations of the Bunch–Kaufman and Aasen’s algorithms were slower than the LU factorization, they preserve the symmetry that can reduce the runtime or memory requirement for the rest of the soft-



(a) Gflop/s



(b) Time (s)

Figure 4. Performance of dense symmetric factorizations (double precision); (a) Gflop/s; (b) time (s).

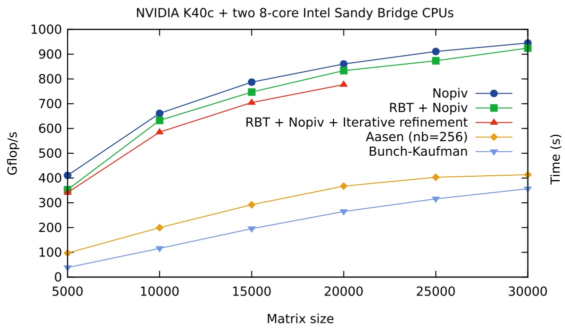
ware (e.g., sparse symmetric factorization, or any simulation code). The symmetric factorization also preserves the inertia of the coefficient matrix.

After having compared the performance of the factorization, we now compare the performance of solving a linear system using random matrices. Figure 5(a) and (b) compares respectively the performance in Gflop/s and time for the symmetric indefinite solvers on multicores with a GPU. The "Gflop/s" is computed as the ratio of the number of flops required for the factorization (i.e., $n^3/3$) plus the number of flops for the solve (i.e., $2n^2 + n$) over time (in seconds). The time for the transfer of the matrices between CPU and GPU is also taken into account. Here, the randomization and the iterative refinement are performed on the GPU; the factorizations are performed with the hybrid CPU/GPU implementations as described previously. The solve is performed on the CPU for Aasen and Bunch–Kaufman and on the GPU for the other implementations. Here, the curve for the RBT solver with iterative refinement stops at size 20,000 because the iterative refinement requires a copy of the original matrix and thereby two times more memory on the GPU. Consistently with the previous experiments, the Aasen solver is slightly faster than the Bunch–Kaufman solver, and the no-pivoting solvers outperform those that use pivoting.

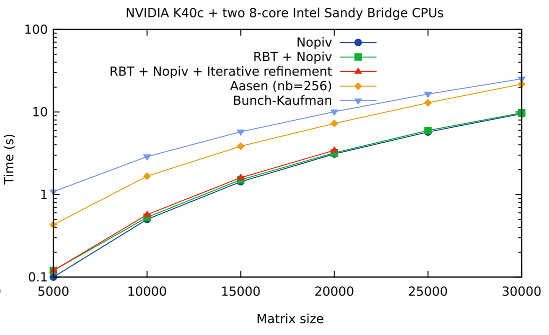
Let us now study the backward error obtained for the linear system solution computed with the corresponding solvers (on random matrices). We plot in Figure 6 the componentwise backward error given in [7, p. 78] and expressed by

$$\omega = \max_i \frac{|Ax - b|_i}{(|A| \cdot |x| + |b|)_i},$$

where x is the computed solution. For the RBT solver, we consider the cases without iterative refinement and with one step of iterative refinement in the working precision. We observe that adding



(a) Gflop/s



(b) Time (s)

Figure 5. Performance of dense symmetric solvers (double precision): (a) flop/s; (b) time (s).

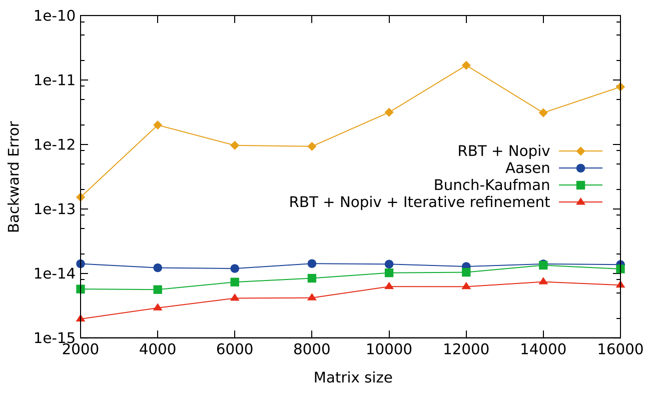


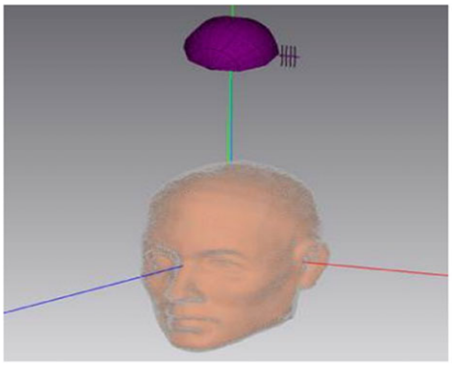
Figure 6. Comparison of componentwise backward error (double precision).

one step of iterative refinement is sufficient to obtain a backward error similar to the other solvers (i.e., in the range $10^{-14} - 10^{-15}$ for the random matrices considered in these experiments).

3.2. Experiments and applications for no-pivoting LDL^T

In some physical applications involving dense symmetric complex non-Hermitian systems, it is not necessary to pivot in the LDL^T factorization (see e.g., [23, p. 209] for more information on this class of matrices). These systems are classically solved using an LU factorization because ScaLAPACK does not provide symmetric factorization for this type of matrix. The application considered here is related to the simulation of processes in which acoustic waves are scattered by obstacles. Unless the geometry of the scattering object is very simple, it is generally not possible to find an analytical solution of scattering problems and then numerical schemes are required. A classical approach is to approximate the solution to time harmonic acoustic problems using the boundary element method (BEM). The BEM discretization leads to linear systems with dense complex symmetric (non-Hermitian) matrices that usually do not require pivoting. Here, we consider two test cases where the scattering objects correspond to a human head and a truck engine (Figure 7).

The matrices (in single complex precision) resulting from the BEM discretization have, respectively, the sizes 10,424 and 15,135. Tables I and II present numerical results for the solution based on our LDL^T factorization with no pivoting on the GPU (see end of Section 2.4, here, no RBT is used), applied to two sample matrices with comparison to LU factorization. Because of the smaller number of flops, our LDL^T factorization enables us to accelerate the calculation by about 48% while keeping a similar accuracy, expressed here by computing the scaled residual $||b - Ax||_\infty/(N||A||_\infty \times ||x||_\infty)$.


(a) Human head

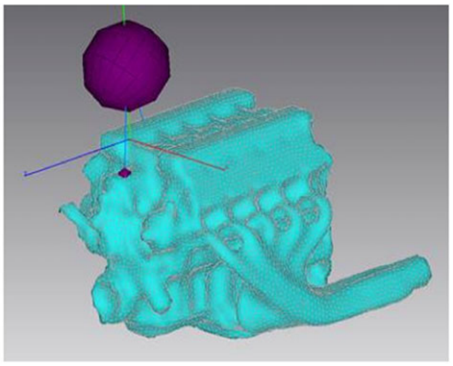

(b) Truck engine

Figure 7. Test cases for acoustic scattering problems: (a) Human head; (b) Truck engine.

Table I. Human head (matrix size is 10,424 in single complex precision).

	Time (s)	Scaled residual
LU	1.34	1.44e-10
LDL^T NoPiv	0.69	1.37e-10

Table II. Car motor (matrix size is 15,135 in single complex precision).

	Time (s)	Scaled residual
LU	3.74	7.46e-11
LDL^T NoPiv	1.93	9.28e-11

3.3. Performance results of mixed-precision iterative refinements

Figure 8 compares the performance of the mixed-precision algorithm (routine ZCHESV) with that of the standard symmetric indefinite solvers using single and double complex precisions (routines CHESV and ZHESV), and on random matrices. We computed the Gflop/s using the flop count needed for the standard algorithm in double complex precision (i.e., $\frac{4}{3}n^3 + 8n^2 n_{rhs} + o(n^2)$ flops needed to compute the LDL^T factorization and to perform a pair of forward and backward substitutions, where n is the dimension of A and n_{rhs} is the number of right-hand sides). The iterative refinement converged in two iterations to obtain the accuracy of the double precision. As we expected, for a large enough matrix, the mixed-precision algorithm obtained a performance close to that in single precision (e.g., for $n = 20,000$, the single precision and mixed-precision solvers were about $1.36\times$ and $1.27\times$ faster than the double precision solver, respectively).

For these experiments, we used the LDL^T factorization without pivoting. This is motivated by our observation that in many real applications, the pivoting is not needed in most of the cases. In the rare case of the failure, the iterative refinement would not converge, signaling the need for pivoting.

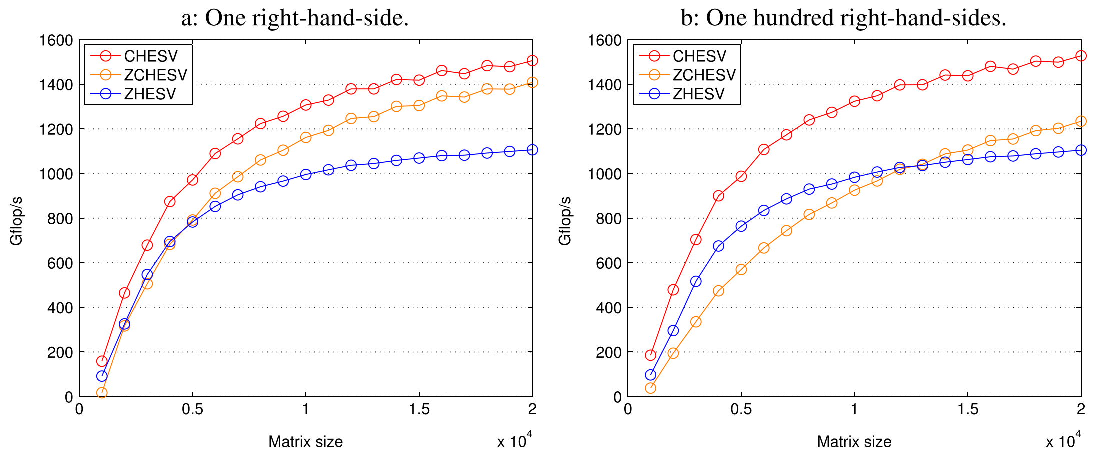


Figure 8. Performance of the standard and mixed-precision solvers: CHESV and ZHESV are the standard solver in single and double complex precision, while ZCHESV is the mixed-precision solver: (a) one right-hand side; (b) one hundred right-hand sides.

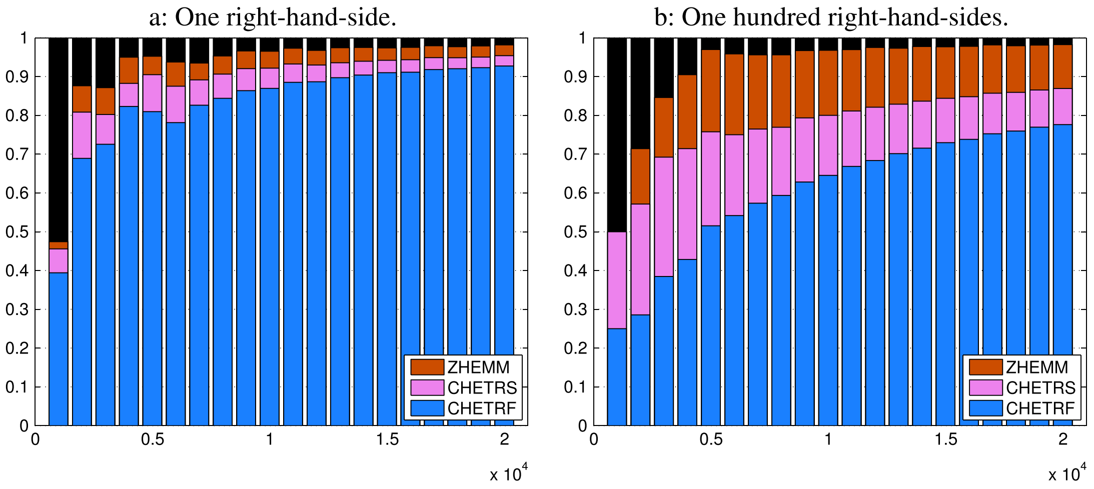


Figure 9. Time breakdown of the mixed-precision solver: CHETRF compute the LDL^T factorization in single complex precision, while CHETRS and ZHEMM are used for iterative refinement to compute the solution with the LDL^T factors and to perform the matrix–matrix multiply, respectively. See Figure 3 for the pseudocode: (a) one right-hand side; (b) one hundred right-hand sides.

When this happens, the user can fall back on a stable algorithm like Bunch and Kaufman's. We can easily integrate the RBT into the mixed-precision solver in order to reduce the probability of encountering small diagonal entries.

In Figure 8, we observe that the performance benefit of using the mixed-precision algorithm decreases as the number of right-hand sides increases. This is due to the increase in the relative overhead associated with the residual computation in double precision compared with the factorization cost. This can be also seen in Figure 9, where the time spent by the double-precision arithmetic increases (e.g., `ZHEMM`).

4. CONCLUSION

We presented the performance of dense symmetric indefinite solvers on hybrid GPU+CPU machines for which until recently, there were no implementations of the algorithms that can utilize the GPU. The symmetric pivoting required to maintain the numerical stability of the factorization leads to frequent synchronizations and exhibits irregular memory accesses, which are difficult to optimize on a GPU. We investigated several techniques to reduce the expensive communication required for pivoting (e.g., native GPU and communication-avoiding implementations). Unfortunately, the overhead associated with the symmetric pivoting can still be significant. However, these algorithms preserve the symmetry, which is required in several physical applications, and reduce the runtime and memory requirement for the rest of the application software. The randomization using RBT followed by an LDL^T factorization without pivoting outperforms other algorithms and is about twice as fast as the LU factorization. We also presented experimental results for acoustic scattering problems where there is no need for pivoting and how mixed precision can be used to enhance performance. Our current implementations are based on standard BLAS/LAPACK routines, and we are improving the performance of factorization by developing specialized GPU kernels. We point out that low-level optimizations are also provided in vendor libraries (e.g., CuSolver implementation of the Bunch–Kaufman algorithm). Our implementations have been released as a part of MAGMA software package, including the iterative refinements which use the mixed-precision arithmetics.

ACKNOWLEDGEMENTS

The authors would like to thank the NSF (grant number ACI-1339822), NVIDIA, and MathWorks for supporting this research effort. The authors are also grateful to Nicolas Zerbib (ESI Group, Compiègne, France) for his help in using test matrices from acoustics.

REFERENCES

1. Björck Å. *Numerical Methods for Least Squares Problems*. SIAM: Philadelphia (USA), 1996.
2. Nédélec J-C. Acoustic and electromagnetic equations. Integral representations for harmonic problems. In *Applied Mathematical Sciences*, Vol. 144. Springer-Verlag: New-York, 2001.
3. Tomov S, Dongarra J, Baboulin M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing* 2010; **36**(5&6):232–240.
4. Bunch JR, Parlett BN. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM Journal on Numerical Analysis* 1971; **8**:639–655.
5. Bunch JR, Kaufman L. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation* 1977; **31**:163–179.
6. Ashcraft C, Grimes RG, Lewis JG. Accurate symmetric indefinite linear equation solvers. *SIAM Journal on Matrix Analysis and Applications* 1998; **20**(2):513–561.
7. Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D. *LAPACK Users' Guide* (3rd Edition). SIAM: Philadelphia (USA), 1999.
8. Aasen J. On the reduction of a symmetric matrix to tridiagonal form. *BIT Numerical Mathematics* 1971; **11**(3): 233–242.
9. Golub GH, Van Loan CF. *Matrix Computations* (Third edition). The Johns Hopkins University Press: Baltimore, 1996.
10. Rozložník M, Shklarski G, Toledo S. Partitioned triangular tridiagonalization. *ACM Transactions on Mathematical Software* 2011; **37**(4):1–16.
11. Blackford L, Choi J, Cleary A, D'Azevedo E, Demmel JW, Dhillon I, Dongarra JJ, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley R. *ScaLAPACK Users Guide*. SIAM: Philadelphia (USA), 1997.

12. Ballard G, Becker D, Demmel J, Dongarra J, Druinsky A, Peled I, Schwartz O, Toledo S, Yamazaki I. Communication-avoiding symmetric-indefinite factorization. *SIAM Journal on Matrix Analysis and Applications* 2014; **35**:1364–1460.
13. Parker DS. Random butterfly transformations with applications in computational linear algebra. *Technical Report CSD-950023*, UCLA Computer Science Department, 1995.
14. Baboulin M, Becker D, Dongarra JJ. A Parallel Tiled Solver for Dense Symmetric Indefinite Systems on Multicore Architectures. *Parallel & Distributed Processing Symposium (IPDPS)*, Shanghai (China), 2012.
15. Baboulin M, Becker D, Bosilca G, Danalis A, Dongarra JJ. An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems. *Parallel Computing* 2014; **40**(7):213–223.
16. Baboulin M, Dongarra J, Rémy A, Tomov S, Yamazaki I. Dense symmetric indefinite factorization on GPU accelerated architectures. *Proceedings of 11th International Conference on Parallel Processing and Applied Mathematics (PPAM 2015)*, Krakow (Poland), 2015; 86–95.
17. Nath R, Tomov S, Dongarra J. An improved MAGMA GEMM for Fermi graphics processing units. *International Journal of High Performance Computing Applications* 2010; **24**(4):511–515.
18. Abalenkovs M, Abdelfattah A, Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Tomov S, Yamazaki I, YarKhan A. Parallel programming models for dense linear algebra on heterogeneous systems. *Supercomputing Frontiers and Innovations* 2015; **2**(4):10–2015.
19. Dongarra J, Kurzak J, Luszczek P, Moore T, Tomov S. *Numerical algorithms and libraries at exascale*. (Available from:http://www.hpcwire.com/2015/10/19/numerical-algorithms-and-libraries-at-exascale/) [October 19 2015. HPCwire].
20. Dongarra J, Gates M, Haidar A, Kurzak J, Luszczek P, Tomov S, Yamazaki I. Accelerating numerical dense linear algebra calculations with GPUs. In *Numerical Computations with GPUs*. Springer International Publishing: Cham (Switzerland), 2014; 1–26.
21. Haidar A, Cao C, Yamazaki I, Dongarra J, Gates M, Luszczek P, Tomov S. Performance and portability with OpenCL for throughput-oriented HPC workloads across accelerators, coprocessors, and multicore processors. *5th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA 14)*, IEEE, New Orleans, LA, 2014; 11-2014.
22. Haidar A, Dongarra J, Kabir K, Gates M, Luszczek P, Tomov S, Jia Y. HPC programming on Intel many-integrated-core hardware with MAGMA port to Xeon Phi. *Scientific Programming* 2015; **23**:01–2015.
23. Higham NJ. *Accuracy and Stability of Numerical Algorithms*. SIAM: Philadelphia (USA), 2002.
24. Intel. Math Kernel Library (MKL). Available from https://software.intel.com/en-us/intel-mkl/.
25. Baboulin M, Dongarra J, Demmel J, Tomov S, Volkov V. Enhancing the performance of dense linear algebra solvers on GPUs in the MAGMA project. *Poster at Supercomputing (SC'08)*, Austin, 2008.
26. Castaldo A, Whaley R. Scaling LAPACK panel operations using parallel cache assignment. *Proceedings of the 15th AGM SIGPLAN Symposium on Principle and Practice of Parallel Programming*, Bangalore (India), 2010; 223–232.
27. Gustavson F. Recursive leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development* 1997; **41**:737–755.
28. Toledo S. Locality of reference in LU decomposition with partial pivoting. *SIAM Journal on Matrix Analysis and Applications* 1997; **18**(4):1065–1081.
29. Demmel J, Grigori L, Hoemmen M, Langou J. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 2012; **34**:A206–A239. also available as EECS Department, University of California, Berkeley, Technical report (UCB/EECS-2008-89).
30. Grigori L, Demmel J, Xiang H. CALU: a communication optimal LU factorization algorithm. *SIAM Journal on Matrix Analysis and Applications* 2011; **32**(4):1317–1350.
31. Abdelfattah A, Haidar A, Tomov S, Dongarra J. Performance, design, and autotuning of batched GEMM for GPUs. *The International Supercomputing Conference (ISC High Performance 2016*, Frankfurt, Germany, 2016.
32. Ballard G, Becker D, Demmel J, Dongarra J, Druinsky A, Peled I, Schwartz O, Toledo S, Yamazaki I. Implementing a blocked Aasen's algorithm with a dynamic scheduler on multicore architectures. *Proceedings of the 27th International Symposium on Parallel and Distributed Processing*, Boston (USA), 2013; 895–907.
33. Yamazaki I, Tomov S, Dongarra J. *Non-GPU-Resident Dense Symmetric Indefinite Factorization, Concurrency and Computation: Practice and Experience*, 2016. doi: 10.1002/cpe.4012.
34. Baboulin M, Dongarra JJ, Hermann J, Tomov S. Accelerating linear system solutions using randomization techniques. *ACM Transactions on Mathematical Software* 2013; **39**(2):1–13.
35. Baboulin M, Li XS, Rouet F-H. Using random butterfly transformations to avoid pivoting in sparse direct methods. *Proceedings of International Conference on Vector and Parallel Processing (VecPar 2014)*, Eugene (OR), USA.
36. Becker D, Baboulin M, Dongarra J. Reducing the amount of pivoting in symmetric indefinite systems. *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011)*, Torun (Poland), 2011; 133–142.
37. Stewart G.W. *Introduction to Matrix Computations*. Academic Press: New York (USA), 1973.
38. Wilkinson JH. *Rounding Errors in Algebraic Processes*. Prentice-Hall: Englewood Cliffs (USA), 1963.
39. Moler CB. Iterative Refinement in Floating Point. *Journal of the ACM* 1967; **14**(2):316–321.

40. Buttari A, Dongarra J, Langou J, Langou J, Luszczek P, Kurzak J. Mixed precision iterative refinement techniques for the solution of dense linear systems. *International Journal of High Performance Computing Applications* 2007; **21**:457–466.
41. Baboulin M, Buttari A, Dongarra J, Kurzak J, Langou J, Luszczek P, Tomov S. Accelerating scientific computations with mixed precision algorithms. *Computer Physics Communications* 2009; **180**(12):2526–2533.
42. Demmel J.W. *Applied Numerical Linear Algebra*. SIAM: Philadelphia (USA), 1997.