Received: 29 May 2016 | Revised: 12 August 2016 | Accepted: 4 September 2016
DOI 10.1002/cpe.4012

WILEY

RESEARCH ARTICLE

# Non-GPU-resident symmetric indefinite factorization

Ichitaro Yamazaki | Stanimire Tomov | Jack Dongarra

Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, USA

**Correspondence**
Ichitaro Yamazaki, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, USA.
Email: iyamazak@eecs.utk.edu

**Funding information**
National Science Foundation NVIDIA Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems, Grant/Award Number: ACI-1339822

## Summary

We study various algorithms to factorize a symmetric indefinite matrix that does not fit in the core memory of a computer. There are two sources of the data movement into the memory: one needed for selecting and applying pivots and the other needed to update each column of the matrix for the factorization. It is a challenge to obtain high performance of such an algorithm when the pivoting is required to ensure the numerical stability of the factorization. For example, when factorizing each column of the matrix, a diagonal entry, which ensures the stability, may need to be selected as a pivot among the remaining diagonals, and moved to the leading diagonal by swapping both the corresponding rows and columns of the matrix. If the pivot is not in the core memory, then it must be loaded into the core memory. For updating the matrix, the data locality may be improved by partitioning the matrix. For example, a right-looking partitioned algorithm first factorizes the leading columns, called panel, and then uses the factorized panel to update the trailing submatrix. This algorithm only accesses the trailing submatrix after each panel factorization (instead of after each column factorization) and performs most of its floating-point operations (flops) using BLAS-3, which can take advantage of the memory hierarchy. However, because the pivots cannot be predetermined, the whole trailing submatrix must be updated before the next panel factorization can start. When the whole submatrix does not fit in the core memory all at once, loading the block columns into the memory can become the performance bottleneck. Similarly, the left-looking variant of the algorithm would require to update each panel with all of the previously factorized columns. This makes it a much greater challenge to implement an efficient out-of-core symmetric indefinite factorization compared with an out-of-core nonsymmetric LU factorization with partial pivoting, which only requires to swap the rows of the matrix and accesses the trailing submatrix after each in-core factorization (instead of after each panel factorization by the symmetric factorization). To reduce the amount of the data transfer, in this paper we uses the recently proposed left-looking communication-avoiding variant of the symmetric factorization algorithm to factorize the columns in the core memory, and then perform the partitioned right-looking out-of-core trailing submatrix updates. This combination may still require to load the pivots into the core memory, but it only updates the trailing submatrix after each in-core factorization, while the previous algorithm updates it after each panel factorization.Although these in-core and out-of-core algorithms can be applied at any level of the memory hierarchy, we apply our designs to the GPU and CPU memory, respectively. We call this specific implementation of the algorithm a non–GPU-resident implementation. Our performance results on the current hybrid CPU/GPU architecture demonstrate that when the matrix is much larger than the GPU memory, the proposed algorithm can obtain significant speedups over the communication-hiding implementations of the previous algorithms.

**KEYWORDS**
dense matrix factorization, non–GPU-resident, out-of-core, symmetric indefinite

  

## 1 | INTRODUCTION

Many scientific and engineering simulations require the solution of a dense symmetric indefinite linear system of equations,

$$A\mathbf{x} = \mathbf{b}, \tag{1}$$

where $A$ is an $n$-by-$n$ dense symmetric indefinite matrix, $\mathbf{b}$ is a given right-hand side, and $\mathbf{x}$ is the solution vector to be computed. To solve the linear system, we first factorize the coefficient matrix $A$ into a product of matrices, eg, $A = LDL^T$, where $L$ is a lower-triangular matrix with unit diagonals and $D$ is a diagonal matrix. Then, to compute the solution $\mathbf{x}$, we solve the corresponding sequence of the linear systems with the respective coefficient matrices $L$, $D$, and $L^T$.

It is a challenge to achieve the high performance of the symmetric indefinite factorization due to the symmetric pivoting that is needed to maintain the numerical stability of the factorization,

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

where $P$ is a permutation matrix representing the row pivots and $D$ is a tridiagonal matrix (including a block diagonal matrix with 1-by-1 or 2-by-2 pivots). The selection of each pivot not only requires synchronizations but also leads to irregular data accesses because only either the upper or lower triangular part of the matrix is stored, and some parts of the pivot column may be stored as the transpose of the corresponding part of the row. Then, the application of the symmetric pivot to the trailing submatrix again requires irregular data accesses.

Because the data access and synchronization have become significantly more expensive compared with the arithmetic operations on the modern computer, the symmetric pivoting can dramatically increase the factorization time. This is also true on a GPU that has become a crucial component in scientific and engineering computation. Nevertheless, with a careful designing and tuning of the implementation, the symmetric indefinite factorization process can be accelerated using the GPU.[1]

Our previous work[1] only considered the cases where the matrix can fit all at once in the GPU memory. As the amount of the GPU memory is limited, this assumption may not hold for many matrices that are of interest.

To address this limitation, in this paper we design and develop non–GPU-resident implementations of the two most popular symmetric indefinite factorization algorithms, Bunch-Kaufman[2] and Aasen's.[3] Because the data transfer through the PCI express can become the bottleneck on the current hardware architectures, our implementations are designed to either hide or avoid such communication. Although the data bandwidth may increase with the future GPU, the relative cost of the data transfer to the computation will most likely increase, and hence, such non–GPU-resident implementations are expected to be more critical for the future GPU. We first implement these two algorithms in a communication-hiding fashion (ie, it aims to hide the communication behind the computation). Such implementations can obtain the speedup of up to 2× over the implementation that does not overlap any communication with the computation.* However, as the communication has become significantly more expensive compared with the computation, our performance results demonstrate that it is a challenge to completely hide the communication behind the computation.

Seeking further acceleration, we develop an implementation that combines the partitioned[4] and communication-avoiding (CA)[5] variants of the Aasen's algorithm; ie, after the GPU-resident factorization in a left-looking CA fashion, the whole trailing submatrix is updated in a right-looking partitioned fashion. Compared with our previous communication-hiding implementations, this implementation significantly reduces the amount of the data traffic between the CPU and the GPU. Namely, our communication-hiding implementations update the trailing submatrix after each panel factorization, while the second implementation updates the submatrix after each in-core factorization.

As a result, it can obtain a great speedup when the matrix is significantly greater than the GPU memory. We note that this implementation is different from the previous CA Aasen's algorithm proposed and studied elsewhere.[1,5,6] Although the previous algorithm avoids some of the communication (instead of hiding the communication), it accesses all the previously factorized column of $L$ for each panel factorization; some of which may not fit in the GPU memory. Hence, its non–GPU-resident implementation would suffer from the excessive data traffic similar to our communication-hiding implementations of the partitioned Bunch-Kaufman or Aasen's algorithm.

There are three main contributions of the paper. First, we present our designs and implementations of the non–GPU-resident partitioned Bunch-Kaufman and Aasen's algorithms in a communication-hiding fashion. We then extend the Aasen's implementation using its CA variant for the GPU-resident factorization. Finally, we study their performance on a current hybrid CPU/GPU architecture. Although we focus on the hybrid architecture in this paper, the current studies may be extended to other architectures (eg, out-of-core implementations).

The rest of the paper is organized as follows. After surveying related work in Section 2, we describe the algorithms that our implementations are based on Section 3. We then, in Section 4, present our non–GPU-resident implementations of the algorithms. Finally, in Sections 5 and 6 we show numerical and performance results, respectively. Final remarks are listed in Section 7.

Throughout this paper, we use $a_{i,j}$ to denote the $(i,j)$th entry of the matrix $A$, 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 $A$. In addition, $\mathbf{a}_{i_1:i_2,j}$ is the column vector consisting of the $i_1$th through the $i_2$th rows of the $j$th column of $A$, while $\mathbf{a}_{i,j_1:j_2}$ is the row vector consisting of the $j_1$th through the $j_2$th column of the $i$th row of $A$. Finally, for our discussion on the block algorithm, we use $A_{i,j}$ and $A_{i_1:i_2,j}$ to denote the $(i,j)$th block and the block column consisting of the $i_1$th through the $i_2$th blocks of the $j$th block column of $A$, respectively, where each block is of dimension $n_b$-by-$n_b$. Finally, $N_b$ is the number of the columns that can fit in the GPU memory at once. All of our experiments were conducted in the 64-bit double precision on 2 eight-core Intel Sandy Bridge CPUs with 1 NVIDIA K20c GPU. The GPU has 11.5 GB of memory, while its double-precision peak performance is 1.43 Tflop/s. The CPU and GPU are connected by a PCIe 3Gen with 16 GB/s. We compiled our code using the GNU gcc version 4.3.4 compiler and the CUDA nvcc version 6.0 compiler with the optimization flag -O3, and linked it with the Intel's threaded Math Kernel Library (MKL) version xe2013.1.046.

*This maximum speedup is obtained when the algorithm spends the equal amount of time on the comutation and communication.

## 2 | RELATED WORK

To ensure the numerical stability of the symmetric indefinite factorization, there are a number of strategies to select the pivots, including the complete pivoting (Bunch-Parlett algorithm),[7] partial pivoting (Bunch-Kaufman algorithm),[2] rook pivoting (bounded Bunch-Kaufman), p. 523[8] fast Bunch-Parlett, p. 525[8] and Aasen's algorithms.[3] In particular, the Bunch-Kaufman and rook pivoting are implemented in LAPACK[9] and are used extensively in many scientific and engineering computations. We are working to integrate the Aasen's algorithm into LAPACK. In this paper, we focus on the 2 representative strategies, the Bunch-Kaufman and Aasen's algorithms.

Most of the previous out-of-core symmetric indefinite factorization algorithms rely on the Bunch-Kaufman algorithm. For instance, the challenges of designing an out-of-core implementation of the algorithm were studies on a distributed-memory CPU computer.[10] There, to reduce the communication overheads and get scalable performance, diagonal pivots were selected within or near the current elimination block. Similarly, out-of-core factorization of a sparse symmetric indefinite matrix based on the Bunch-Kaufman algorithm was considered.[11] For reduction of the cost of selecting the pivot, the *delayed* scheme was employed where the columns that cannot be stably factorized within a block are moved to the trailing submatrix, requiring a dynamic data structure to accommodate the delayed columns.

Unlike these previous approaches, our implementation is based on the Aasen's algorithm, and it globally searches for the pivots and can be implemented using a simple static data structure.

Non–GPU-resident implementations of LU, QR, and Cholesky factorization are described in Yamazaki et al.[12] Unlike the symmetric indefinite factorization (which requires the whole trailing submatrix to be updated after each panel factorization), the panel factorization of these *one-sided* factorizations only requires the next panel to be updated. Hence, their non–GPU-resident factorization can be simply implemented in the left-looking fashion.

Although our focus of this paper is on the deterministic algorithms with theoretical error bounds, there are growing interests in randomized algorithms.[13] When combined with the iterative refinements, these randomized algorithms may compute the solution of the desired accuracy without pivoting, while obtaining the high performance on modern computers.[1,14,15]

## 3 | ALGORITHMS

In this section, we discuss the 3 symmetric indefinite factorization algorithms, the partitioned Bunch-Kaufman,[2,16] the partitioned Aasen's,[3,4] and the CA Aasen's[5], which are studied in this paper.

### 3.1 | Partitioned Bunch-Kaufman

Let us assume that the first $j-1$ columns of the matrices $L$ and $D$ of the $LDL^T$ factorization Equation 2 have been computed. Then, the $j$th columns of the matrices are computed by first updating the $j$th column $\mathbf{a}_j$ of $A$ using the previous columns,

$$\mathbf{w}_{j:n,j} := \mathbf{a}_{j:n,j} - L_{j:n,1:j-1} D_{1:j-1,1:j-1} \ell^T_{j,1:j-1}, \tag{3}$$

and then computing

$$\ell_{j:n,j} := \frac{\mathbf{w}_{j:n,j}}{w_{j,j}} \quad \text{and} \quad d_{j,j} := w_{j,j}. \tag{4}$$

This process is repeated for $j = 1, 2, \ldots, n$ to factorize the whole matrix $A$ and is referred to as a *left-looking* formulation of the algorithm because, at each step, the column $\mathbf{a}_j$ is updated with the previous columns, which are on the left of $\mathbf{a}_j$.

The above algorithm is based on BLAS-1 and BLAS-2, which allow only a small number of data reuses. Because the data access has become expensive, they obtain only a fraction of the performance of BLAS-3, which can exploit more data reuses. To take advantage of the memory hierarchy using BLAS-3, after a fixed number $n_b$ of the columns are factorized, a *partitioned* variant of the Bunch-Kaufman algorithm updates the trailing submatrix $A^{(2,2)}$ by

$$A^{(2,2)} := A^{(2,2)} - L^{(2,1)} D^{(1,1)} (L^{(2,1)})^T, \tag{5}$$

where $A^{(1,1)} = A_{1:n_b,1:n_b}$ and $A^{(2,2)} = A_{n_b+1:n,n_b+1:n}$, and the off-diagonal block $L^{(2,1)}$ is defined, accordingly; ie, the matrix $A$ is partitioned as

$$A = \left( \begin{array}{c|c} A^{(1,1)} & A^{(2,1)T} \\ \hline A^{(2,1)} & A^{(2,2)} \end{array} \right)$$

and factorized as

$$\left( \begin{array}{c|c} L^{(1,1)} & \\ \hline L^{(2,1)} & I \end{array} \right) \left( \begin{array}{c|c} D^{(1,1)} & \\ \hline & A^{(2,2)} \end{array} \right) \left( \begin{array}{c|c} L^{(1,1)T} & L^{(2,1)T} \\ \hline & I \end{array} \right).$$

Then, the same procedure is applied to the trailing submatrix $A^{(2,2)}$. The current set of the $n_b$ columns being factorized is referred to as a *panel*, and this algorithm is referred to as *right-looking* becaus the panel is used to update the trailing submatrix, which is on the right of the panel.

A numerical issue comes when the column $\mathbf{w}_{j:n,j}$ is scaled by a scalar $w_{j,j}$ of small magnitude in Equation 4. For numerical stability to be maintained while keeping the symmetry of the matrix, the Bunch-Kaufman algorithm selects either 1-by-1 or 2-by-2 pivots from the diagonals of the remaining submatrix. Figure 1A shows the pseudocode of this partitioned algorithm that is implemented in LAPACK.[16] It performs the same number of flops as the column-wise algorithm (ie, $\frac{1}{3}n^3 + O(n^2)$ flops) and is backward stable subject to a growth factor. The selection of the pivot requires the $r$th column and row of $A$. Because the diagonal pivot $a_{r,r}$ cannot be determined until the $j$th step of the factorization, the non–GPU-resident factorization must update the whole trailing submatrix after each panel factorization.

### 3.2 | Partitioned Aasen's Algorithm

For the symmetric indefinite linear system of Equation 1 to be solved, the Aasen's algorithm[3] computes the $LTL^T$ factorization of $A$,

$$A = LTL^T,$$

```
$\alpha = (1+\sqrt{17})/8$
$j = 1$
while $j < n$ do
    $k = j$
    {Panel factorization}
    while $j < k + n_b - 1$ do
        Update $\mathbf{a}_j$ with the previous columns
        $r = \arg\max_{i>j} |a_{i,j}|$ and $\omega_1 = |a_{r,j}|$
        if $\omega_1 > 0$ then
            if $|a_{j,j}| \geq \alpha\omega_1$ then
                $s = 1$
                Use $a_{j,j}$ as a $1 \times 1$ pivot.
            else
                Update $\mathbf{a}_r$ with the previous columns

                $\omega_r = \max_{i \geq j; i \neq r} |a_{i,r}|$
                if $|a_{j,j}|\omega_r \geq \alpha\omega_1^2$ then
                    $s = 1$
                    Use $a_{j,j}$ as a $1 \times 1$ pivot.
                else
                    if $|a_{r,r}| \geq \alpha\omega_r$ 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 extract
        $\boldsymbol{\ell}_{j:j+s-1}$
        $j = j + s$
    end while
    {Trailing submatrix update}
    $A^{(2,2)} := A^{(2,2)} - L^{(2,1)} D^{(1,1)} (L^{(2,1)})^T$
end while
```

(A)

```
for $J = 0, n_b, 2n_b, \ldots, n$ do
    for $j = J, J+1, \ldots, J + n_b - 1$ do
        $\mathbf{h}_{j+1:n,j+1} := \mathbf{a}_{j+1:n,j+1}$
                $- H_{j:n,1:j} \boldsymbol{\ell}_{j+1,1:j}^T$
        $\mathbf{w} := \mathbf{h}_{j+1:n,j+1}$
        if $j > 0$ then
            $\mathbf{w} := \mathbf{w} - \boldsymbol{\ell}_{j+1:n,j} t_{j,j+1}$
        end if
        $t_{j+1,j+1} := w_1$
        if $j < n - 1$ then
            $\mathbf{v} := \mathbf{w}_{2:n} \boldsymbol{\ell}_{j+2:n,j+1}$
            $k = \arg\min |v|_i$
            swap $i$-th and $k$-th rows of $L$ and $H$
            swap $i$-th and $k$-th rows and columns of $A$
            $t_{j+2,j+1} = v_1$
            $\boldsymbol{\ell}_{j+1:n,j+1} = \mathbf{v} / t_{j+2,j+1}$
        end if
    end for
    $A^{(2,2)} := A^{(2,2)} - H^{(2,1)} (L^{(2,1)})^T$
            $- \boldsymbol{\ell}_{j+1:n,j} t_{j+1,j} \boldsymbol{\ell}_{j+1:n,j+1}^T$
end for
```

(B)

```
for $j = 1, 2, \ldots, n_t$ do
    for $i = 2, 3, \ldots, j - 1$ do
        $X := T_{i,i-1} L_{j,i-1}^T$
        $Y := T_{i,i} L_{j,i}^T$
        $Z := T_{i,i+1} L_{j,i+1}^T$
        $W_{i,j} := 0.5Y + Z$
        $H_{j,i}^T := X + Y + Z$
    end for
    $C := A_{j,j} - L_{j,2:j-1} W_{2:j-1,j}$
            $- W_{2:j-1,j}^T L_{j,2:j-1}^T$
    $T_{j,j} := L_{j,j}^{-1} C L_{j,j}^{-T}$
    if $j < \frac{n}{n_b}$ then
        if $j > 1$ then
            $H_{j,j}^T := T_{j,j-1} L_{j,j-1}^T + T_{j,j} L_{j,j}^T$
        end if
        $E := A_{j+1:n_t,j} - L_{j+1:n,2:j} H_{j,2:j}^T$
        $[L_{j+1:n_t,j+1}, H_{j,j+1}^T, P^{(j)}] := \mathrm{LU}(E)$

        $T_{j+1,j} := H_{j,j+1}^T L_{j,j}^{-T}$

        $L_{j+1:n_t,2:j} := P^{(j)} L_{j+1:n_t,2:j}$
        $A_{j+1:n_t,j+1:n_t} := P^{(j)} A_{j+1:n_t,j+1:n_t} P^{(j)T}$

        $P_{j+1:n_t,1:n_t} := P^{(j)} P_{j+1:n_t,1:n_t}$
    end if
end for
```

(C)

**FIGURE 1** Pseudocodes of symmetric indefinite factorization algorithms. A, Bunch-Kaufman algorithm. B, Partitioned Aasen's algorithm, where $H = LT$ and $\ell 1$ is the first column of the identity matrix. C, Communication-avoiding Aasen's algorithm, where $HT = TLT$ and $L_{:,1}$ is the first $n_b$ columns of the identity matrix and $n_t$ is the number of block columns in A, ie, $nt = \frac{n}{n_b}$

where $L$ is still lower-triangular with unit diagonals but $T$ is now symmetric tridiagonal. For the memory hierarchy on a modern computer to be exploited, a partitioned variant of the Aasen's algorithm[4] first factorizes a panel in a left-looking fashion, using an intermediate Hessenberg matrix $H$ that is defined as $H = LT$. Namely, we first set the first column $\boldsymbol{\ell}_1$ of $L$ to be the first column of an identity matrix. Then, for $j = 1, 2, \ldots, n$, assuming that the first $(j-1)$ columns of $H$ and the first $j$ columns of $L$ have been computed, the $j$th column of $H$ is computed from the $j$th column of the equation $A = HL^T$,

$$\mathbf{h}_{j:n,j} \ell_{j,j}^T := \mathbf{a}_{j:n,j} - H_{j:n,1:j-1} \ell_{j,1:j-1}^T,$$

where $\ell_{j,j}$ is a unit diagonal. Also, from the $j$th column of the equation $H = LT$, we have

$$\mathbf{h}_{j:n,j} = \ell_{j:n,j-1} t_{j-1,j} + \ell_{j:n,j} t_{j,j} + \ell_{j:n,j+1} t_{j+1,j}.$$

Hence, if we let $\mathbf{w} = \boldsymbol{\ell}_{j:n,j} t_{j,j} + \boldsymbol{\ell}_{j:n,j+1} t_{j+1,j}$, then we can compute it by

$$\mathbf{w} := \mathbf{h}_{j:n,j} - \ell_{j:n,j-1} t_{j-1,j},$$

and because $w_1 = \ell_{j,j} t_{j,j} + \ell_{j,j+1} t_{j+1,j}$, and $\ell_{j,j}$ is 1 and $\ell_{j,j+1}$ is 0, we have

$$t_{j,j} := w_1.$$

Finally, because $\mathbf{w}_{2:n} = \boldsymbol{\ell}_{j+1:n,j} t_{j,j} + \boldsymbol{\ell}_{j+1:n,j+1} t_{j+1,j}$, the $(j+1)$-th column of $L$ can be computed by

$$\ell_{j+1:n,j+1} := \frac{\mathbf{v}}{v_1} \quad \text{and} \quad t_{j+1,j} := v_1,$$

where $\mathbf{v} = \mathbf{w}_{2:n} - \boldsymbol{\ell}_{j+1:n,j} t_{j,j}$. For the stability to be maintained, the element with the largest module in $\mathbf{v}$ is used as the pivot.

After the left-looking panel factorization, the trailing submatrix is updated in a right-looking fashion,

$$A^{(2,2)} := A^{(2,2)} - H^{(2,1)} (L^{(2,1)})^T - \ell_{n_b}^{(2,1)} t_{n_b+1,n_b} (\ell_1^{(2,2)})^T, \tag{6}$$

where the matrix is partitioned as in Equation 5 and $\ell_{n_b}^{(2,1)}$ and $\ell_1^{(2,2)}$ are the last and first columns of $L^{(2,1)}$ and $L^{(2,2)}$, respectively. Then, the same procedure is recursively applied on the trailing submatrix $A^{(2,2)}$. Figure 1B shows the pseudocode of the algorithm.

In comparison with a standard column-wise algorithm, this partitioned algorithm requires an additional rank-1 update of the trailing submatrix, performing about $\frac{1}{3}n_t n^2$ additional flops, where $n_t$ is the number of block columns (ie, $n_t = \frac{n}{n_b}$).†

However, BLAS-3 can be used to perform most of these flops,

and it has been shown that this partitioned algorithm can shorten the factorization time on modern computers.[4]

Unlike the Bunch-Kaufman, the Aasen's only uses the $j$th column of $A$ to select the $j$th pivot. However, this pivot column becomes the next column to be factorized and can come from anywhere in the trailing submatrix. Hence, like the Bunch-Kaufman, the non–GPU-resident implementation of this partitioned Aasen's must update the whole trailing submatrix after each panel factorization.

### 3.3 | Communication-avoiding Aasen's algorithm

Recently, a CA variant of the Aasen's algorithm was developed by replacing all the element-wise operations with block-wise operations.[5] For avoidance of storing the whole matrix $H$, it computes the block row of $H$ at each step, which is then discarded.

Namely, from the $j$th block column of $H^T = TL^T$ and the $(j, j)$th block of $A = LH^T$, we have for $k = 1, 2, \ldots, j-1$,

$$H_{j,k}^T = T_{k,k-1}L_{j,k-1}^T + T_{k,k}L_{j,k}^T + T_{k,k+1}L_{j,k+1}^T \tag{7}$$

and

$$H_{j,j}^T = L_{j,j}^{-1}\left(A_{j,j} - \sum_{k=1}^{j} L_{j,k}H_{j,k}^T\right). \tag{8}$$

Then, we obtain $T_{j,j}$ from the $(j, j)$th block of $H^T = TL^T$:

$$T_{j,j} = \left(H_{j,j}^T - T_{j,j-1}L_{j,j-1}^T\right)L_{j,j}^{-T}. \tag{9}$$

Unfortunately, the above procedure is unstable because the symmetric $T_{j,j}$ is computed through a sequence of unsymmetric expressions. To recover the symmetry, we substitute $H_{j,j}^T$ of Equation 8 and $H_{j,k}^T$ of Equation 7 into Equation 9 and compute $T_{j,j}$ as in

$$L_{j,j}T_{j,j}L_{j,j}^T = A_{j,j} - \sum_{k=1}^{j-1} L_{j,k}W_{j,k} - \sum_{k=1}^{j-1} W_{j,k}^T L_{j,k}^T,$$

where $W_{j,k} = \frac{1}{2}U_{j,k} + V_{j,k}$, $U_{j,k} = T_{k,k}L_{j,k}^T$, and $V_{j,k} = T_{k,k+1}L_{j,k+1}^T$. Finally, from the $(j, j)$th block of $H^T = TL^T$, we compute $H_{j,j}^T$ by

$$H_{j,j}^T = T_{j,j}L_{j,j}^T + T_{j,j-1}L_{j,j-1}^T.$$

Next, from the $j$th block column of $A = LH^T$, we can extract the $(j + 1)$th block column of $L$,

$$P_j^T L_{(j+1):n,j+1}H_{j,j+1}^T = \mathrm{LU}(V), \tag{10}$$

where

$$V = A_{(j+1):n,j} - \sum_{k=1}^{j} L_{(j+1):n,k}H_{j,k}^T,$$

and $L_{(j+1):n,j+1}$ and $H_{j,j+1}^T$ are the L and U factors of $V$ with the partial pivoting $P_j$. This partial pivoting is then applied to the corresponding part of the submatrices, ie,

$$A_{j+1:n,j+1} := P_j A_{(j+1):n,(j+1):n} P_j^T$$

and

$$L_{(j+1):n,1:j} := P_j L_{(j+1):n,1:j}.$$

Finally, from the $(j + 1, j)$th block of $H = TL^T$, we have

$$T_{j+1,j} = H_{j,j+1}^T L_{j,j}^{-T}.$$

Figure 1C shows the pseudocode of this CA Aasen's algorithm that performs the same number of flops, $\frac{1}{3}n^3 + O(n^2)$, as the Bunch-Kaufman algorithm.‡ This CA algorithm updated each panel using all the previous block columns in the left-looking fashion. Hence, to factorize each panel, its non–GPU-resident implementation must read all the previous block columns into the GPU memory.

## 4 | IMPLEMENTATIONS

We now describe our non–GPU-resident implementations to factorize the symmetric indefinite matrix based on the algorithms in Section 3.

### 4.1 | Partitioned Bunch-Kaufman

A challenge of implementing the Bunch-Kaufman algorithm in a non–GPU-resident fashion is that, at each step of the panel factorization, the diagonal pivot may be selected from anywhere in the trailing submatrix. Hence, although at each step, each pivot column is updated by the previously factorized columns on the GPU (ie, left-looking), we may need to transfer the pivot column from the CPU. In addition, after the panel factorization, before the next panel factorization can start, the whole trailing submatrix must be updated, where some parts of the submatrix are not on the GPU. This distinguishes the non–GPU-resident Bunch-Kaufman factorization from the non–GPU-resident LU, QR, or Cholesky factorization, which can be simply implemented in a left-looking fashion (after the submatrix on the GPU is updated using the previously factorized block columns, the GPU-resident factorization assesses only the block columns on the GPU). In this section, we describe our non–GPU-resident implementation of the partitioned Bunch-Kaufman algorithm that aims to hide the communication behind the computation.

#### 4.1.1 | FIFO cache to store trailing submatrix.

For the trailing submatrix update, we update the lower-triangular part of the submatrix one block column at a time. Since all the block columns do not fit in the GPU memory at once, we manually manage a First-In-First-Out (FIFO) cache to store these block columns in the GPU memory, and create a lookup table to associate each block column to the location in the cache (see Figure 2 for an illustration). In addition, if a remaining block column that needs to be updated is still on the CPU, then as soon as the next block column is updated, we copy it back to the CPU and prefetch the remaining block column into the GPU. In this way, we aim to hide the communication of the block column behind the update of the block columns that are already on the GPU. We use multiple GPU streams and events not only to overlap the communication with the computation but also to update multiple block

†The right-looking variant of the Aasen's algorithm, the Parlett-Reid algorithm,[17] performs twice more flops than the left-looking column-wise Aasen's algorithm.

‡We referred to this as a CA algorithm although our implementation does not use a CA algorithm for the panel factorization.

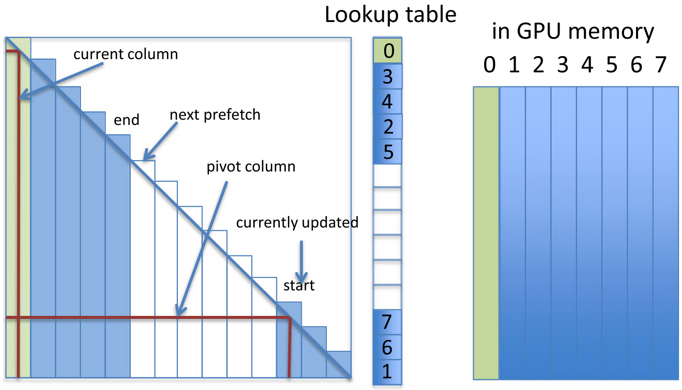


**FIGURE 2** Non–GPU-resident partitioned factorization. Only the lower block triangular part of the matrix is stored in the GPU memory. The dark lines show the current and pivot columns that must be swapped

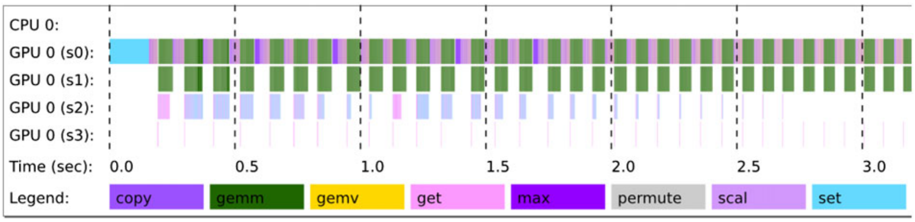


**FIGURE 3** Execution trace of non–GPU-resident Bunch-Kaufman factorization, where 2 GPU streams are used for updating the trailing submatrix and 2 other GPU streams are used for communication

columns in parallel (relying on the CUDA runtime to schedule the independent tasks and obtain the load balance) if the block size is too small to utilize the compute power of the GPU for updating the single block column (see Figure 3). Because, at each step of the panel factorization, the previous columns of the panel are used to update the pivot column, we keep the panel in the cache all the time. Once the panel factorization and submatrix update are completed, we copy the panel back to the CPU. If the next panel is in the cache, we copy it to the designated cache location. Otherwise, the panel is copied from the CPU before the panel factorization starts.

The transfer of the block columns from the CPU to the GPU is pipelined such that the transfer of all the block columns on the CPU is overlapped with the whole trailing submatrix update. For this, we have 2 parameters $n_b$ and $N_b$ to be tuned: the parameter $n_b$ specifies the block or panel size and $N_b$ is the number of the columns that can fit in the GPU memory at once and hence specifies the number of the columns for the GPU-resident factorization (in our performance studies, we used $n_b = O(100)$ and $N_b = O(1000)$).[§] When updating the trailing submatrix after each panel factorization, the block size $n_b$ affects the performance of the matrix-matrix multiply to update each block column, but it does not significantly affect the total time to transfer the block columns to the GPU. On the other hand, the parameter $N_b$ determines how much of the data transfer can be hidden behind the computation. However, the trailing submatrix is accessed after each panel factorization, and hence, the total amount of the data transfer for factorizing the matrix $A$ is proportional to the total number of panels, $\frac{n}{n_b}$. Although the amount of the data transfer is minimized by setting the panel size to be $n_b = N_b$, this makes the GPU-resident factorization perform all of its computation using BLAS-1 and BLAS-2. We study the effects of the parameters $n_b$ and $N_b$ to the performance of the non–GPU-resident factorization in Section 6.

### 4.1.2 | Non–GPU-resident symmetric pivoting.

At each step of the panel factorization, we first copy the current column $\mathbf{a}_j$ into a GPU memory workspace and update it with the previous columns of the panel, all of which are on the GPU. Then, based on the numerical values of the entries in the column, we select a candidate for the pivot column, which can be anywhere in the trailing submatrix. This candidate column is copied into another workspace and updated using the previous columns. Once the pivot columns are updated, they are copied as the factorized columns, while the original columns may be copied into the trailing submatrix.

For the non–GPU-resident implementation, if the pivot column is on the CPU, then it must be copied to the GPU before being updated with the previous columns. In addition, only the lower-triangular part of the submatrix is stored, and some parts of the pivot column may be stored as the transpose of the corresponding part of the row, some parts of which can be on the CPU (see Figure 2). This irregular access to the columns and rows on the GPU and CPU makes it difficult to obtain good performance of the panel factorization. In addition, as Figure 2 shows, although the contiguous block columns are stored in our FIFO cache, the block columns may not be stored contiguously in our cache. For copying the pivot column from the corresponding row of the multiple

[§] Is it possible to use a different block size from the panel size or 2D block layout for updating the trailing submatrix. However, the amount of the data reuse is determined by the panel size $n_b$, and hence, our implementation uses the block size $n_b$ and 1D layout for the update.

block columns on the GPU, we use a batched GPU kernel that executes the multiple copy operations on the multiple block columns in a single GPU kernel launch. Then, the remaining parts of the pivot column are copied from the CPU.

#### 4.1.3 | Shifting after 2-by-2 pivot.

An additional complication arises when the algorithm selects a 2-by-2 pivot for the last column of the panel factorization. When this happens, we push the last column to the next panel, decreasing the size of the current panel by 1 column. To adjust the block column boundaries, we first copy back the last column of the last block column in the GPU memory to the CPU (ie, the last column of the block column labeled as "end" in Figure 2). Then, we shift the remaining columns in the block column to the right and copy the last column from the previous block column as the new first column of the block. This process is recursively applied to the previous block columns until the first column of the first block column is copied from the CPU (ie, the first column of the block column labeled as "start" in Figure 2).

#### 4.1.4 | GPU memory workspace.

Since $W^{(2,1)} = L^{(2,1)}D^{(1,1)}$, where $W$ is computed in Equation 3, we use a workspace memory to store $W_{:,1:n_b}$ such that we avoid recomputing $W_{:,1:n_b}$ for the trailing submatrix update Equation 5. Hence, we require the workspace of dimension $n$-by-$(n_b + 1)$ with an extra column to store the potential 2-by-2 pivot for the last column of the panel.

#### 4.1.5 | Alternative implementation

The left-looking Bunch-Kaufman can bring in and factorize 1 column at a time on the GPU, and once no more column can fit in the GPU memory, we can update the trailing submatrix (ie, $n_b = N_b$). This allows us to bring in the trailing submatrix into the GPU memory for each GPU-resident factorization. Unfortunately, this performs the GPU-resident factorization using the BLAS-1- and BLAS-2-based panel factorization. In our experiments, this implementation obtained only a fraction of the above BLAS-3-based partitioned implementations of the algorithm.

### 4.2 | Partitioned Aasen's

To select a pivot, beside the current column, which is on the GPU, Bunch-Kaufman may access an additional column $\mathbf{a}_r$ which could be on the CPU. On the other hand, Aasen's algorithm selects the pivot based only on the numerical entries of the current column, which is already on the GPU. Hence, for the pivot selection, the non–GPU-resident implementation of the Aasen's algorithm does not transfer data between the CPU and the GPU. However, once the pivot is selected, both Bunch-Kaufman and Aasen's algorithms symmetrically pivot the trailing submatrix. Hence, the Aasen's may need to copy the column from the CPU, while the Bunch-Kaufman has already copied the pivot column (although the data copy is wasted if the second candidate is not selected as a pivot by the Bunch-Kaufman).[¶] Fortunately, unlike Bunch-Kaufman, the Aasen's algorithm does not use 2-by-2 pivots. Hence, the panel size stays the same throughout the factorization.

Like Bunch-Kaufman, the panel factorization of the Aasen's algorithm may select the pivots from anywhere in the trailing submatrix. Hence, the Aasen's algorithm updates the whole trailing submatrix before the next panel factorization. For the remaining block columns on the GPU to be stored, our non–GPU-resident implementation of the Aasen's algorithm uses the same FIFO cache as that used for Bunch-Kaufman. For the trailing submatrix update Equation 6, we merge the rank-1 and rank-$n_d$ updates into a single rank-$(n_d + 1)$ update such that it can be performed by a single BLAS-3 call.

Because the first column of $L$ is the first column of the identity matrix and the diagonals of $L$ are ones, they are not stored. Hence, we store the $j$th column $\ell_{j+1:n,j}$ of $L$ in the $(j - 1)$th column $\mathbf{a}_{j:n,j-1}$ of $A$.

Then, the tridiagonal matrix $T$ can be stored in the main diagonal of $A$ and the first off-diagonal below them (ie, $t_{j,j}$ and $t_{j+1,j}$ are stored in $a_{j,j}$ and $a_{j+1,j}$, respectively). Finally, our implementation uses an $n$-by-$(1 + n_b)$ memory workspace to store $H^{(2,1)}$ and $\ell^{(2,1)}_{n_b} t_{n_b+1,n_b}$ for the trailing submatrix update Equation 6.

### 4.3 | Partitioned communication-avoiding Aasen's

Both the right-looking partitioned Bunch-Kaufman and Aasen's algorithms update the whole trailing submatrix after each panel factorization. Similarly, the left-looking CA Aasen's algorithm needs to access all the previous block columns for updating each panel. When the block columns do not fit in the GPU memory all at once, although the communication is overlapped with the computation as much as possible, the data transfer between the CPU and the GPU becomes overwhelmingly expensive. As a result, the data transfer cannot be completely hidden behind the computation, and the performance of the non–GPU-resident factorization suffers. To reduce the amount of the data transfer, in this section we perform the right-looking update of the whole trailing submatrix after each GPU-resident factorization by the CA Aasen's algorithm. To distinguish it from our 2 previous implementations, we refer to this new implementation as our non–GPU-resident implementation of the partitioned CA Aasen's algorithm.

Figure 4 shows the pseudocode of our implementation, where $n_b$ is the block size and $N_b$ is the number of columns of $A$ that can fit in the GPU memory at once. At each step of the GPU-resident factorization, the next panel is copied from the CPU, while the symmetric pivoting is applied on the fly. Because the CA Aasen's algorithm updates the panel in the left-looking fashion, the trailing submatrix does not have to be updated after the panel factorization. The factorized panels are kept on the GPU such that they can be used to update the remaining block columns. This GPU-resident factorization continues until no more panel can fit in the GPU memory. Although the CA algorithm used for the GPU-resident factorization performs most of the flops using BLAS-3, most of these operations are on the blocks of dimension $n_b$. To efficiently utilize the GPU, we use GPU streams and events extensively to exploit the parallelism between the small BLAS-3 calls.

Once the GPU-resident factorization is completed, the remaining block columns of the trailing submatrix are copied to the GPU 1 block column at a time, updated using all the factorized block columns in the

[¶]The Aasen's algorithm swaps trailing submatrix $A_{j+1:n,j+1}$ based on the $j$th pivot. Hence, the last pivot of the panel factorization swaps $\mathbf{a}_{n_d+1}$, which is not in the panel and can be on the CPU.

**for** $J = 1, 1 + n_t, \ldots, N_t$ **do**
  {GPU-resident CA factorization}
  **for** $j = J, J+1, \ldots, J + nt - 1$ **do**
    Use CA-Aasen's to compute $L_{:,j}$ and $T_{:,j}$ on GPU
    Copy $A_{:,j+1}$ from CPU after symmetric pivot
  **end for**
  {Non-GPU-resident submatrix update}
  **for** $j = J + n_t, J + n_t + 1, \ldots, N_t$ **do**
    Copy $A_{:,j}$ from CPU, and compute $H_{J:J+n_t-1,j}$
    $A_{j:N_t,j} := A_{j:N_t,j} - L_{j:N_t,J:J+n_t-1} H_{J:J+n_t-1,j}$
      $- L_{j:N_t,J+n_t}(T_{J+nt,J+nt-1} L_{j,J+n_t-1})$
    Copy $A_j$ back to CPU
  **end for**
  Copy $A_{:J+n_t}$ to GPU
**end for**

**FIGURE 4** Partitioned communication-avoiding Aasen's algorithm, where $n_t = \frac{N_b}{n_b}$ and $N_t = \frac{n}{n_b}$

GPU memory, and copied back to the CPU. To avoid storing $H^{(2,1)}$, unlike the partitioned Aasen's algorithm, our implementation follows the CA Aasen's algorithm and updates the trailing submatrix by

$$A^{(2,2)} = A^{(2,2)} - L^{(2,1)}(H^{(2,1)})^T - L^{(2,1)}_{n_t}(T_{n_b+1,n_b}(L^{(2,2)}_1)^T).$$

To update the block column of $A^{(2,2)}$, we compute the corresponding block column of $(H^{(2,1)})^T$ and then discard it such that we can reuse the memory workspace to store the next block column. While our previous implementation of the partitioned Aasen's algorithm combined 2 low-rank updates of the trailing submatrix into 1, our implementation of the partitioned CA Aasen's algorithm uses multiple GPU streams and events to utilize GPU (overlapping the data transfer of the remaining block column with the updating of other columns and updating the multiple block columns in parallel). Once all the trailing block columns are updated, we copied all the block columns on the GPU back to the CPU, except for the next panel.

Any stable LU algorithm can be used for the panel factorization Equation 10 (eg, partial[18] or tournament[19] pivoting). Our implementation can also compute the LU factorization on the GPU or CPU (eg, using MKL). Although the CPU is often efficient performing the BLAS-1– and BLAS-2–based panel factorization, this requires to copy back the LU factors to the GPU. However, because the factors are already on the CPU, it avoids the needs to copy them back after the trailing submatrix.

This algorithm performs an extra rank-$n_b$ update of the trailing submatrix, requiring $\frac{1}{3}N_t n_b n^2$ more flops than the column-wise Aasen's algorithm. However, unlike previous algorithms, the trailing submatrix is updated only after each GPU-resident factorization, while the in-core factorization accesses only the block columns on the GPU and uses BLAS-3 calls. As a result, this algorithm can significantly reduce the amount of the data traffic between the CPU and the GPU.

In addition, compared with the previous implementations, this implementation has more regular data accesses and its implementation can be significantly simpler.

Our implementation uses 2 $n$-by-$(n_s n_b)$ workspaces where $n_s$ is the number of the GPU streams used for the update; one to store $H_{:,1:n_b}$, and the other to store the block columns of the trailing submatrix. Beside these 2 workspaces, we allocate 4 $n$-by-$n_b$ workspaces to store the auxiliary matrices $X$, $Y$, $Z$, and $W$.

## 5 | NUMERICAL RESULTS

Although, in this paper, we focus on studying the performance of the non–GPU-resident implementations, we have conducted extensive numerical experiments to compare the numerical behaviors of various Bunch–Kaufman and Aasen's algorithms. For the sake of completeness, Figure 5 shows the results with random matrices, which are representative of many other results. We used the LU factorization with partial pivoting to solve the banded linear system for the CA Aasen's algorithm, while the Givens QR factorization is used to solve the tridiagonal system of the partitioned Aasen's algorithm as in Rozloz'ník et al.[4] Although the Aasen's algorithm obtained slightly greater backward errors, the forward and backward errors of all the standard Bunch-Kaufman and Aasen's algorithms were in the same order. The rook pivoting[8] avoids the potential numerical issues associated with the Bunch-Kaufman algorithm because of the large growth factor in $L$, but the figure indicates that, for these random matrices, the Bunch-Kaufman was as stable as the rook pivoting. On the other hand, the backward errors of the CA Aasen's algorithm were about an order of magnitude greater than the standard algorithms. As explained elsewhere,[5,6] this is expected because the backward errors of the CA Aasen's algorithm depend linearly to the block size (ie, $n_d = 128$). A few iterations of iterative refinement can smooth out the residual norm. The figure also indicates that our implementation that combines the CA Aasen's algorithm with the partitioned right-looking trailing submatrix update obtained about the same numerical errors as the CA Aasen's algorithm.

## 6 | PERFORMANCE RESULTS

Finally, we study the performance of our 3 non–GPU-resident implementations using a fixed amount of the GPU memory.[‖] For the partitioned CA Aasen's, we factorized the panel on the CPU using threaded MKL.

Figure 6 shows the breakdown of the symmetric indefinite factorization time. The black part of each bar is mostly the time needed to transfer the block columns of the trailing submatrix between the CPU and the GPU during the trailing submatrix update. Clearly, the proposed algorithm spends much less time moving the data between the CPU and GPU. This is not only because the proposed algorithm moves a smaller amount of data, but also because while the other implementations only update each trailing block column using the $n_b$ column, the proposed algorithm updates each block column with the $N_b$ columns, better hiding the data transfer behind the computation. The figures also show that compared with the other implementations, the CA Aasen's may spend shorter time in the matrix-matrix multiply (ie, GEMM). This is because while the other algorithms update each block column using one of the previous block columns at a time (right-looking), the CA algorithm updates each block column using all of the previous block columns in the core memory at once (left-looking). Finally, in this figure, we see that, compared with the partitioned

‖ The GPU memory is also used for the workspace.

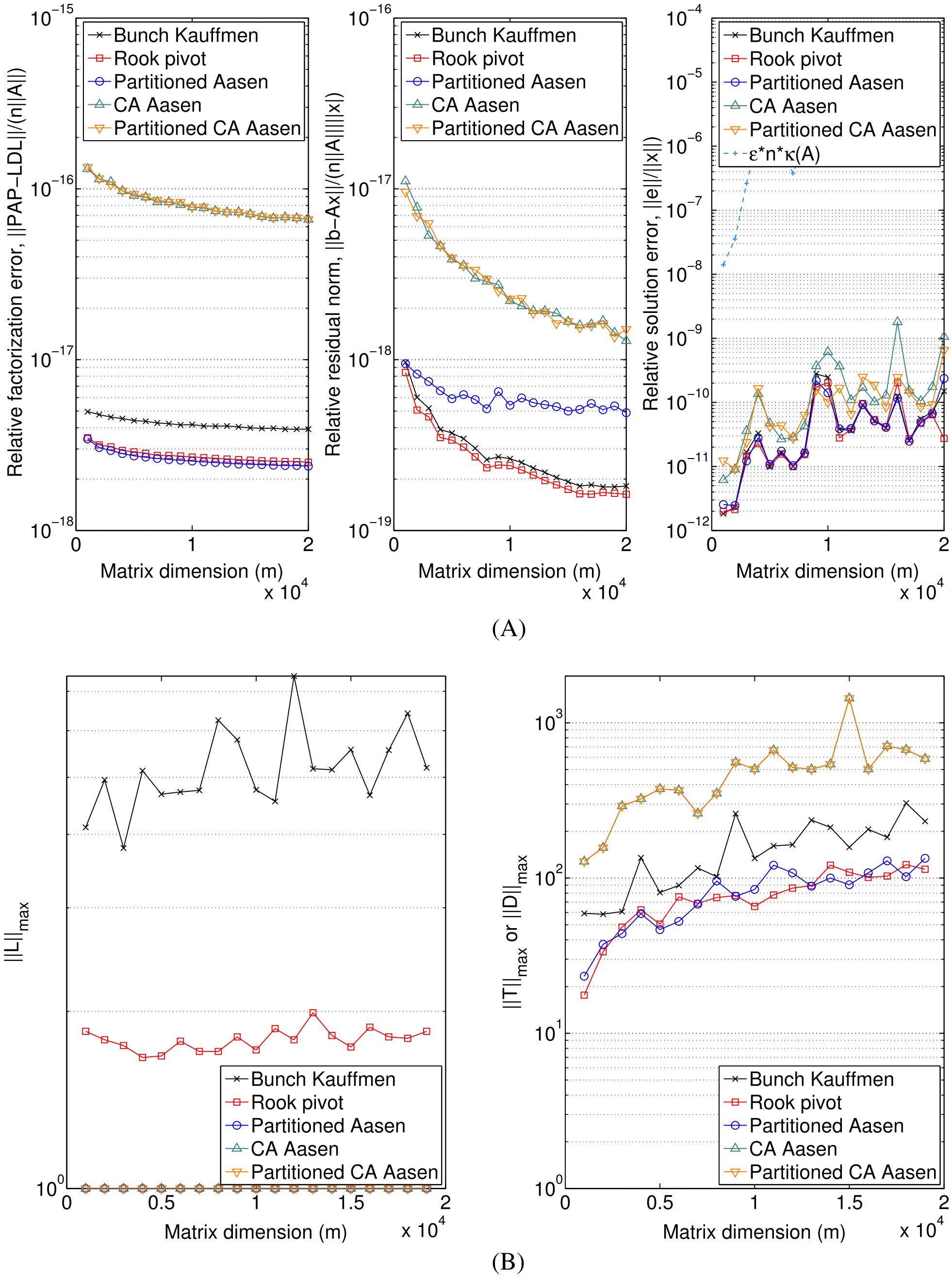


**FIGURE 5** Numerical errors of different symmetric indefinite solvers ($n_b = 128$ and $N_b = 640$). A, Error norms and, B, factor norms

Bunch-Kaufman, the partitioned Aasen's algorithm spent more time in pivoting. This is because once the panel is used to update the trailing submatrix, the right-looking Bunch-Kaufman algorithm does not use the panel for the rest of the factorization. Hence, like its LAPACK implementation, we do not apply the pivot to the previous columns of the matrix *L*. On the other, we have not integrated this optimization into our implementation of the partitioned Aasen's algorithm. The CA Aasen's algorithm updates the panel in left-looking fashion. Hence, our partitioned CA Aasen's algorithm must apply the pivoting to all the previous block columns in the GPU memory. Our implementation applies the pivots to all the previous columns to keep the solver simple.[**]

Figure 7 compares the performance of our 3 non–GPU-resident factorization algorithms and the GPU-resident Bunch-Kaufman algorithms, which are developed for the matrices that fit in the GPU memory (that labeled "Hybrid" performs the panel factorization on the CPU, while that without the label performs the whole factorization on the GPU). These GPU-resident routines are used for the previous studies[1] and are in the latest release of MAGMA software package.[††] The figure clearly demonstrates that the partitioned CA Aasen's obtained significant speedups when the matrix was significantly larger than the available GPU memory. In Figure 7B, we also show the performance of the Bunch-Kaufman algorithm when the communication is not overlapped

[**]The number of pivots may be reduced by applying a matrix ordering (eg, Duff and Pralet[20]) before the numerical factorization.

[††]http://icl.utk.edu/magma/.

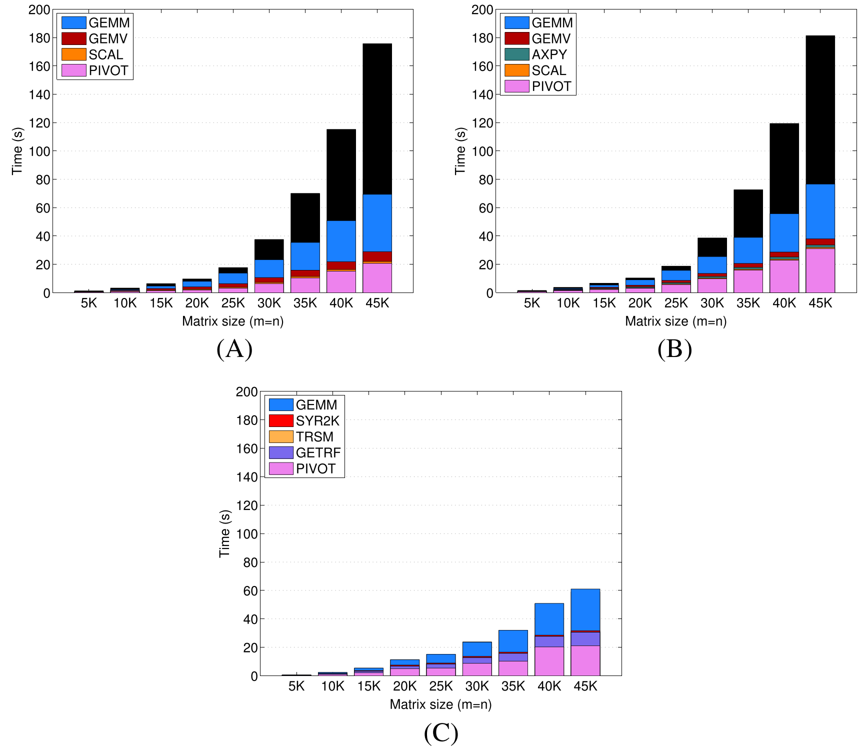


**FIGURE 6** Breakdown of factorization time using 30% of total GPU memory ($n_b = 128$). The whole matrix fits in the GPU memory at once when $n$ is less than 25K (ie, $N_b \geq 25\,000$). The black parts of the bars show the rest of the factorization time that is mostly the time needed to transfer the block columns of the trailing submatrix between the CPU and the GPU during the trailing submatrix update. A, Partitioned Bunch-Kaufman. B, Partitioned Aasen's. C, Partitioned CA Aasen's

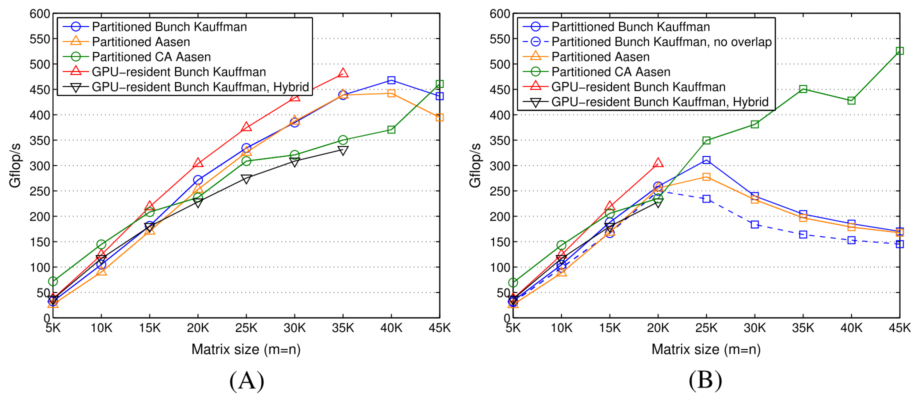


**FIGURE 7** Performance of factorization algorithms ($n_b = 128$) for random matrices, where circle or square markers indicate that the matrices are small enough to fit in the GPU memory, or they are too large to fit in the memory, respectively; ie, using 90% or 30% of the GPU memory, the matrix does not fit in the GPU memory when its dimension is greater than 35 000 and 20 000, respectively (ie, $N_b \approx 35\,000$ and 20 000). The triangle markers are for the in-core factorization. A, Using 90% of total GPU memory and, B, using 30% of total GPU memory

with the computation. We see a smaller gain in the performance as the matrix size grows, indicating that there is not enough computation to hide the communication. In the end, compared with the partitioned Bunch-Kaufman, the partitioned CA Aasen's obtained the speedups of about 1.1× to 3.1× when 30% of the GPU memory was used.

When we increase the block size $n_b$, the partitioned algorithms update the trailing submatrix with larger block columns, improving its performance. However, a larger block size also makes the panel factorization more expensive.

Because the panel factorization is based on BLAS-1 and BLAS-2, it often obtains only a small fraction of the peak performance and could become the performance bottleneck, especially when the whole matrix fit on the GPU. On the other hand, while the partitioned algorithms operate on block columns, the CA Aasen's operates on blocks. In general, even though we use the GPU streams for the CA algorithm to exploit the parallelism, for the GPU to be efficiently utilized the CA Aasen's algorithm requires a larger block size than the partitioned algorithms. For our experiments so far, the block size $n_b$ is set to be 128 for all the algorithms, which obtained good performance of the GPU-resident partitioned algorithms. However, the performance of the CA Aasen's could be improved using a larger block size. Now, for the trailing submatrix update of the non–GPU-resident factorization, the parameter

$N_b$ determines how much of the data transfer can be hidden behind the computation, while the block size $n_b$ affects the performance of the matrix-matrix multiply to update each block column but does not significantly affects the time to transfer the trailing block columns. On the other hand, unlike the partitioned CA implementation that accesses the trailing submatrix before each in-core factorization, the partitioned algorithms access the trailing submatrix after each panel factorization. As a result, the total amount of the data transfer depends on the panel size, and their performance can be improved using a larger block size $n_b$. Figure 8 shows the performance using different block sizes. Although the larger block size improved the performance of the partitioned algorithms, the CA algorithm still shows the performance advantage. In addition, if the block size is too large, then the performance of the GPU-resident partitioned factorization suffers.

Figure 8B also compares the performance of our partitioned CA Aasen's implementation when the panel is factorized on the CPU or on the GPU. When the GPU is used to factorize the panel, we can avoid transferring the factored panel back to the GPU. However, for the panel factorization to utilize the GPU well, it must be implemented carefully because the CPU is efficient in performing the BLAS-1– and BLAS-2–based panel factorization. With our implementation, performing the panel factorization on the GPU only slightly improved the performance when the matrix is large enough.

To compute the solution of the linear system of Equation 1, our partitioned CA Aasen's implementation would require to solve the linear system with the banded matrix $T$. Although we have not implemented the banded solver on the GPU, for reference, Figure 9A shows the fraction of the time spent on the general banded solver of MKL over the time spent by our CA Aasen's factorization. A larger block size reduces the factorization time, while increasing the solve time. Although the fraction of the solve time increases with the block size, it is only a small overhead compared with the factorization time. Finally, our partitioned CA algorithm may provide an additional tuning parameter $N_b$ to improve the performance of the GPU-resident Assen's algorithms.

For instance, compared with the right-looking algorithms, the left-looking CA Aasen's algorithm exhibits smaller parallelism, and it requires a careful implementation to obtain high performance.[6] By periodically performing the right-looking updates, the performance of the CA Aasen's algorithm may be improved (eg, in Figure 7, the performance of the CA algorithm was higher using 30% of the GPU memory than using 90% of the memory). Figure 9B demonstrates this potential by showing the performance of the partitioned CA algorithm, where the matrix fits in the GPU memory, but the trailing submatrix is periodically updated.

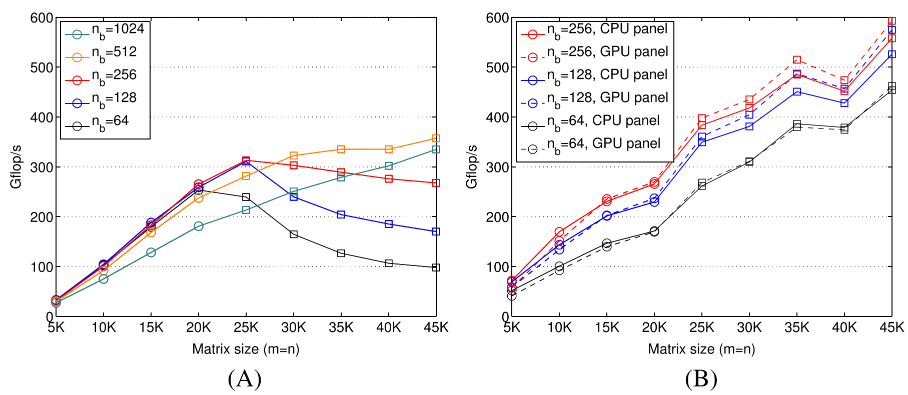


**FIGURE 8** Performance of partitioned Bunch-Kaufman and CA Aasen's factorization for random matrices, using different block sizes and 30% of GPU memory. A, Partitioned Bunch Kaufman and, B, Partitioned CA Aasen's. CA indicates communication-avoiding

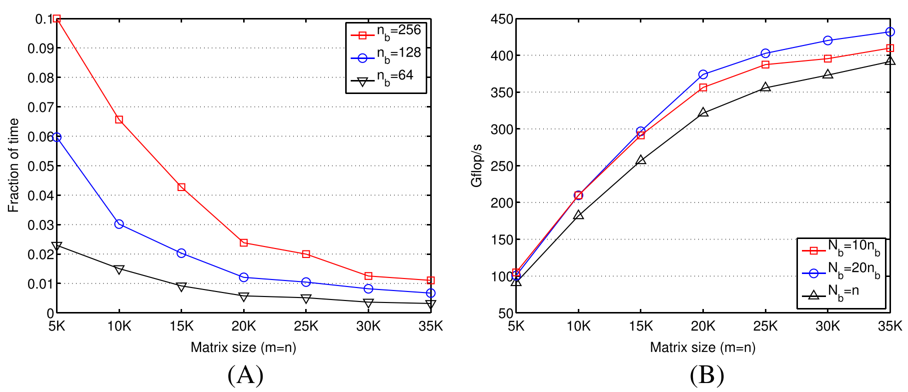


**FIGURE 9** Performance statistics of partitioned CA Aasen's algorithm. A, Fraction of time spent by MKL general banded solver over partitioned CA Aasen's factorization and, B, GPU-resident performance with different values of $N_b$. CA indicates communication-avoiding; MKL, Math Kernel Library

## 7 | CONCLUSION

We designed an out-of-core implementation of the Aasen's algorithm to factorize a symmetric indefinite matrix. Our implementation uses a CA variant of the left-looking algorithm for in-core factorization and then updates the trailing submatrix only after each in-core factorization. Unlike our communication-hiding implementations of the Bunch-Kaufman and Aasen's algorithms that access the trailing submatrix after each panel factorization, this new implementation significantly reduces the data traffic into and out of the core memory. Although the backward errors depend linearly to the block size and could be slightly greater, the performance results of our particular implementations of these algorithms on the current hybrid CPU/GPU architecture demonstrated that the new implementation can obtain significant speedups over the previous implementations when the matrix is significantly larger than the available GPU memory.

We are also interested if it is possible to implement our algorithm on other architectures.

### ACKNOWLEDGMENTS

We thank Alex Druinsky and Jim Demmel for helpful discussions and Gil Shklarski and Sivan Toledo for sharing their implementation of the partitioned Aasen's algorithm. This research was supported in part by the National Science Foundation under grant no. ACI-1339822, NVIDIA, and "Matrix Algebra for GPU and Multicore Architectures (MAGMA) for Large Petascale Systems."

### REFERENCES

1. Baboulin M, Dongarra J, Rémy A, Tomov S, Yamazaki I. Dense symmetric indefinite factorization on GPU accelerated architectures. *The Proceedings of the International Conference on Parallel Processing and Applied Mathematics (PPAM)*, Krakow, Poland; 2015.
2. Bunch J, Kaufman L. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*. 1977;31(137):163–179.
3. Aasen J. On the reduction of a symmetric matrix to tridiagonal form. *BIT*. 1971;11:233–242.
4. Rozloz'ník M, Shklarski G, Toledo S. Partitioned triangular tridiagonalization. *ACM Trans Math Softw*. 2011;37(4):1–16.
5. Ballard G, Becker D, Demmel J, Dongarra J, Druinsky A, Peled I, Schwartz O, Toledo S, Yamazaki I. A communication avoiding symmetric indefinite factorization. *SIAM J Matrix Anal Appl*. 2014;35(4):1364–1406.
6. 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 IEEE International Symposium on Parallel and Distributed Processing*, Boston, Massachusetts, USA; 2013:895–907.
7. Bunch JR, Parlett BN. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J Numerical Anal*. 1971;8:639–655.
8. Ashcraft C, Grimes R, Lewis J. Accurate symmetric indefinite linear equation solvers. *SIAM J Matrix Anal Appl*. 1998;20:513–561.
9. Anderson E, Bai Z, Dongarra J, Greenbaum A, McKenney JDCA, Hammarling S, Demmel J, Bischof C, Sorensen D. Lapack: a portable linear algebra library for high-performance computers. *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, New York, New York, USA; 1990.
10. Strazdins P. Issues in the design of scalable out-of-core dense symmetric indefinite factorization algorithms. *Lecture Notes in Computer Science*. 2003;2659:715–724.
11. Meshar O, Irony D, Toledo S. An out-of-core sparse symmetric-indefinite factorization method. *ACM Trans Math Softw*. 2006;32(3):445–471.
12. Yamazaki I, Tomov S, Dongarra J. One-sided dense matrix factorizations on a multicore with multiple GPU accelerators*. *Procedia Computer Science 9 (2012) 37–46, Proceedings of the International Conference on Computational Science*. ICCS, Omaha, Nebraska; 2012.
13. Parker D. *Random Butterfly Transformations With Applications in Computational Linear Algebra, Tech. Rep. CSD-950023*. Los Angeles: University of California; 1995.
14. Baboulin M, Becker D, Dongarra J. A parallel tiled solver for dense symmetric indefinite systems on multicore architectures. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*; 2013;14–15.
15. Baboulin M, Becker D, Bosilica G, Danalis A, Dongarra J. An efficient distributed randomized solver with application to large dense linear systems. *Parallel Computing*. 2014;40:213–223.
16. Anderson E, Dongarra J. Evaluating block algorithm variants in LAPACK. *SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, Illinois, USA; 1989:3–8.
17. Parlett B, Reid J. On the solution of a system of linear equations whose matrix is symmetric but not definite. *BIT*. 1970;10:386–397.
18. Trefethen L, Scheiber R. Average-case stability of Gaussian elimination. *SIAM J Matrix Anal Appl*. 1990;11(3):335–360.
19. Grigori L, Demmel J, Xiang H. Communication avoiding Gaussian elimination. *The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC08)*, Austin, Texas; 2008:29:1–29:12.
20. Duff I, Pralet S. Strategies for scaling and pivoting for sparse symmetric indefinite problems. *SIAM J Matrix Anal Appl*. 2005;27(2):313–340.

**How to cite this article:** Yamazaki I, Tomov S, Dongarra J. Non-GPU-resident symmetric indefinite factorization. *Concurrency Computat.: Pract. Exper*.2017;29(5):e4012. https://doi.org/10.1002/cpe.4012