Dense Symmetric Indefinite Factorization on GPU Accelerated Architectures

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

[1] University of Paris-Sud and Inria, France
[baboulin,aremy]@lri.fr
[2] University of Tennessee, Knoxville, USA
[dongarra,tomov,iyamazak]@eecs.utk.edu

Abstract. We study the performance of dense symmetric indefinite factorizations (Bunch-Kaufman and Aasen's algorithms) on multicore CPUs with a Graphics Processing Unit (GPU). Though such algorithms are needed in many scientific and engineering simulations, obtaining high performance of the factorization on the GPU is difficult because the pivoting that is required to ensure the numerical stability of the factorization leads to frequent synchronizations and irregular data accesses. As a result, until recently, there has not been any implementation of these algorithms on hybrid CPU/GPU architectures. To improve their performance on the hybrid architecture, we explore different techniques to reduce the expensive communication and synchronization between the CPU and GPU, or on the GPU. We also study the performance of an LDL^T factorization with no pivoting combined with the preprocessing technique based on Random Butterfly Transformations. Though such transformations only have probabilistic results on the numerical stability, they avoid the pivoting and obtain a great performance on the GPU.

Keywords: dense symmetric indefinite factorization, communication-avoiding, randomization, GPU computation.

1 Introduction

A symmetric matrix A is called indefinite when its quadratic form x^TAx 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 [15, 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 [21]. 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 due to their high-performance capabilities. To achieve the performance, however, the algorithms must be designed for high parallelism, high flops to data ratio, and be architecture-aware. A dense symmetric indefinite solver which can efficiently use the GPU's high computing power could lead to new discoveries in the field of physics. The use of the GPU is also motivated by its low energy consumption. For example, a single K40 NVIDIA GPU has a double precision peak of $1,689$ Gflop/s for a thermal design power (TDP) of 235 W. Optimized large dense matrix computations, e.g., matrix-matrix multiplications, reach $1,200$ Gflop/s for a power draw of about 200 W, i.e., ≈ 6 Gflop/W. In contrast, two Sandy Bridge E5-2670 CPUs have about the same TDP ($2 \times 115 = 230$ W) as the K40 but for a peak of 333 Gflop/s, which translates to only 1.4 Gflop/W.

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 triangular and block-diagonal systems. The pivoting strategies to compute the permutation matrix P for the LDL^T factorization include complete pivoting (Bunch-Parlett algorithm) [11], partial pivoting (Bunch-Kaufman algorithm) [12], rook pivoting (bounded Bunch-Kaufman) [4, p. 523], and fast Bunch-Parlett [4, p. 525]. In particular, the Bunch-Kaufman and rook pivoting are implemented in LAPACK [2], a set of dense linear algebra routines on multicore CPUs that are used extensively 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.

Another promising method for solving a symmetric indefinite linear system is the Aasen's method [1], 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 left-looking formulation of the algorithm requires about the same number of floating point operations (flops) that are required to compute the LDL^T factorization. A block algorithm for computing the LTL^T factorization was also proposed [23]. Though the block implementation performs slightly more flops, it can exploit a modern's computer memory hierarchy and obtain performance similar to the Bunch-Kaufman algorithm implemented in LAPACK.

To maintain numerical stability, the pivoting techniques mentioned above involve between $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$ comparisons to search for pivots and possible interchanges of selected columns and rows. This leads to synchronization and data movement at each step of the factorization, which have become significantly more expensive compared to the arithmetic operations on modern computers.

Furthermore, due to the symmetric storage used to store A, the symmetric pivoting requires irregular data access. This increases dramatically the cost of the data movement, making it difficult to obtain the higher performance of the symmetric indefinite factorization. Recently, a communication-avoiding variant of the Aasen's algorithm was proposed [9], which can compute the factorization with a minimum amount of communication. However, the pivoting must still be applied symmetrically, leading to irregular data access. Due to these performance challenges, ScaLAPACK [10], 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 [3]. 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 new state-of-the-art outlook for this important problem. Another technique studied in this paper is a symmetric version of Random Butterfly Transformations (RBT) [22] 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 has been studied on multicore systems [8] and distributed-memory systems [6], but its performance has not been investigated on a GPU.

This paper is organized as follows. Section 2 describes three methods for solving dense symmetric indefinite systems (Bunch-Kaufman and Aasen's algorithms, random butterfly transformations) and their implementation for hybrid CPU/GPU architectures. Section 3 presents performance results for the factorizations using the above algorithms and how they compare with the LU factorization. Section 4 contains concluding remarks.

2 Symmetric Indefinite Factorizations with a GPU

2.1 Bunch-Kaufman Algorithm

The most widely used algorithm for solving a symmetric indefinite linear system is based on the block LDL^T factorization with the Bunch-Kaufman algorithm [12], which is also implemented in LAPACK [2]. The pseudo-code of the algorithm is shown in Figure 1a. To select the pivot at each step of the factorization, it scans 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 is backward stable, subject to the growth factor [20, p. 219]. Then a variant of the Bunch-Kaufman algorithm, also called "rook pivoting", was proposed in [4] 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, each of which requires expensive synchronization. Hence, in this paper, we focus on the Bunch-Kaufman algorithm as a baseline for our performance comparison.

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

```
$\alpha = (1+\sqrt{17})/8, \quad k = 1$
while $k < n$ do
    $\omega_1 = \max_{i>k} |a_{ik}| := |a_{rk}|$
    if $\omega_1 > 0$ then
        if $|a_{kk}| \geq \alpha\omega_1$ then
            $s = 1$
            Use $a_{kk}$ as a $1 \times 1$ pivot.
        else
            $\omega_r = \max_{i \geq k; i \neq r} |a_{ir}|$
            if $|a_{kk}|\omega_r \geq \alpha\omega_1^2$ then
                $s = 1$
                Use $a_{kk}$ as a $1 \times 1$ pivot.
            else
                if $|a_{rr}| \geq \alpha\omega_r$ then
                    $s = 1$
                    Swap rows/columns $(k, r)$
                    Use $a_{rr}$ as a $1 \times 1$ pivot.
                else
                    $s = 2$
                    Swap rows/columns $(k+1, r)$
                    Use $\begin{pmatrix} a_{kk} & a_{rk} \\ a_{rk} & a_{rr} \end{pmatrix}$ as $2{\times}2$ pivot.
                end if
            end if
        end if
    else
        $s = 1$
    end if
    $k = k + s$
end while
```

(a) Bunch-Kaufman.

```
for $j = 1, 2, \dots, \frac{n}{n_b}$ do
    for $i = 2, 3, \dots, j-1$ do
        $X = T_{i,i-1} L_{j,i-1}^T$
        $Y = T_{i,i} L_{j,i}^T$
        $Z = T_{i,i+1} L_{j,i+1}^T$
        $W_{i,j} = 0.5Y + Z$
        $H_{i,j} = X + Y + Z$
    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 $j < n$ then
        if $j > 1$ then
            $H_{j,j} = T_{j,j-1} L_{j,j-1}^T + T_{j,j} L_{j,j}^T$
        end if

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

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

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

(b) Communication-avoiding Aasen's.

Fig. 1: Symmetric indefinite factorization algorithm.

Our first implementation of the Bunch-Kaufman algorithm is based on a hybrid CPU/GPU programming paradigm where the block column (commonly referred to as the *panel*) is factorized on the CPU (e.g., using the multithreaded MKL library [3]), 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 exhibits high data parallelism and can be efficiently implemented on the GPU [24]. Unfortunately, at each step of the panel factorization, the Bunch-Kaufman algorithm may select the pivot from the trailing submatrix. Hence, though 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. Though the 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.

When the entire factorization is implemented on the GPU, up to two columns of the trailing submatrix must be scanned to select a pivot at each step of the the Bunch-Kaufman algorithm – 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 since 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.

2.2 Aasen's Algorithm

To solve a symmetric indefinite linear system, Aasen's algorithm [1] factorizes A into an LTL^T decomposition. The left-looking algorithm takes advantage of the symmetry of A and performs $\frac{1}{3}n^3 + O(n^2)$ flops, which is 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 as the pivot, leading to more regular data access compared to the Bunch-Kaufman algorithm. To exploit the memory hierarchy of modern computers, a blocked version of the algorithm was developed [23], which is based on a left-looking panel factorization, followed by a right-looking trailing submatrix update using BLAS-3 routines. Compared to 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. 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 [9]. 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 [16, 19, 25] or communication-avoiding LU (TSLU, for the panel) [17, 18]. 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 1b.

The GPU has a greater memory bandwidth than the CPU, but the memory accesses are still expensive compared to the arithmetic operations. Hence, our implementation is based on the CA Aasen's algorithm. Though this algorithm performs most of the flops using BLAS-3 (e.g., xGEMM), most of the operations are on the submatrices of the block size n_b. In order to exploit parallelism between the small BLAS calls, we use GPU streams extensively. In addition, for an efficient application of the symmetric pivots after the panel factorization, we apply the pivots in two steps. The first step copies all the columns of the trailing submatrix, which need 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. 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 same pivoting strategy is used to exploit the parallelism on multicore CPU [13]. We tested using the LU factorization with partial pivoting as the panel factorization, using either the multithreaded MKL library on the CPU or using its native GPU implementation in MAGMA on the GPU. Though 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.

2.3 Random Butterfly Transformations

Random Butterfly Transformation (RBT) is a randomization technique initially described by Parker [22] and recently revisited for dense linear systems, either general [5] or symmetric indefinite [6]. It has also been applied recently to a sparse direct solver in a preliminary paper [7]. 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 [14] where it is shown that in practice, $d = 1$ or 2 gives satisfactory results (possibly using a few steps of iterative refinement). 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 [8].

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 [5]. 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

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 $U^Tb$, solve $A_r y = U^T b$, then $x = Uy$
```

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.

3 Experimental Results

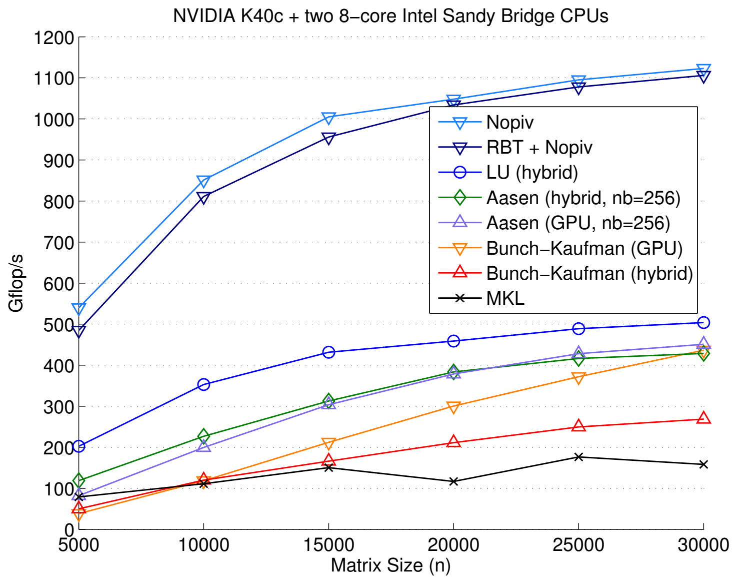

Fig. 2: Performance of symmetric factorization (double precision).

Figure 2 compares the performance of the symmetric indefinite factorizations on multicores with a GPU, where 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 in [8, 14], it may not be numerically stable for some matrices. However, the performance of all the symmetric factorizations with provable stability was lower than that of the LU factorization, demonstrating the cost of the irregular data access associated with the symmetric storage. In addition, though our current implementations of the Bunch-Kaufman and Aasen's algorithms were slower than the LU factorization, they preserve the symmetry which can reduce the runtime or memory requirement for the rest of the software (e.g., sparse symmetric factorization, or any simulation code).

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., [20, p. 209] for more information on this class of matrices). These systems are classically solved using an LU factorization since ScaLAPACK does not provide symmetric factorization for this type of matrix. Here we consider test matrices (in single complex precision) discretized by the boundary element method, used to approximate the solution of harmonic acoustic problems. Tables 1 and 2 present numerical results for the solution based on our LDL^T factorization with no pivoting on the GPU (see Section 2.3), applied to two sample matrices with comparison to LU factorization. Due to 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 $||b - Ax||_\infty/(N||A||_\infty \times ||x||_\infty)$.

4 Conclusion

We presented the performance of symmetric indefinite factorizations on multicore CPUs accelerated by a GPU. The symmetric pivoting required to maintain

Table 1: Human head (matrix size is $10,424$ in single complex precision).

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

Table 2: Car motor (matrix size is $15,135$ in single complex precision).

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

the numerical stability of the factorization leads to frequent synchronizations and exhibits irregular memory accesses which are difficult to optimize on a GPU. As a result, until recently, there were no implementations of the algorithms that can utilize the GPU. To enhance performance, 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 reduces the runtime and memory requirement for the rest of the application software. Though it only has a probabilistic error bound, randomization using RBT followed by an LDL^T factorization without pivoting outperforms other algorithms, and is about twice as fast as the LU factorization. Our current implementations are based on standard BLAS/LAPACK routines, and we are improving the performance of factorization by developing specialized GPU kernels. These implementations will be integrated in a new release of MAGMA.

Acknowledgments

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

References

1. J. Aasen, *On the reduction of a symmetric matrix to tridiagonal form*, BIT **11** (1971), 233–242.
2. E. Anderson, Z. Bai, J. J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. W. Demmel, C. Bischof, D. Sorensen, *LAPACK: a portable linear algebra library for high-performance computers*, Proceedings of the 1990 ACM/IEEE conference on Supercomputing.
3. Intel, *Math Kernel Library (MKL)*, http://www.intel.com/software/products/mkl/.
4. C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. and Appl.*, 20(2):513–561, 1998.
5. M. Baboulin, J. J. Dongarra, J. Hermann, and S. Tomov, *Accelerating Linear System Solutions using Randomization Techniques*, ACM Transactions on Mathematical Software, 39(2), 2013.
6. M. Baboulin, D. Becker, G. Bosilca, A. Danalis, and J. J. Dongarra, *An efficient distributed randomized algorithm for solving large dense symmetric indefinite linear systems*, Parallel Computing, 40(7):213–223, 2014.

7. M. Baboulin, X. S. Li, and F-H. Rouet, *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.
8. M. Baboulin, D. Becker, and J. J. Dongarra, *A Parallel Tiled Solver for Dense Symmetric Indefinite Systems on Multicore Architectures*, Parallel & Distributed Processing Symposium (IPDPS), 2012.
9. G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, and I. Yamazaki, *Communication-avoiding symmetric-indefinite factorization*, SIAM J. Matrix Anal. Appl. **35** (2014), 1364–1460.
10. L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. W. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. Whaley. *ScaLAPACK Users Guide*, SIAM, 1997.
11. J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *SIAM J. Numerical Analysis*, 8:639–655, 1971.
12. J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Math. Comput.*, 31:163–179, 1977.
13. G. Ballard, D. Becker, J. Demmel, J. Dongarra, A. Druinsky, I. Peled, O. Schwartz, S. Toledo, and I. Yamazaki, *Implementing a blocked Aasen's algorithm with a dynamic scheduler on multicore architectures*, Proceedings of the 27th international symposium on parallel and distributed processing, 2013, pp. 895–907.
14. D. Becker, M. Baboulin, and J. J. Dongarra, *Reducing the amount of pivoting in symmetric indefinite systems*, Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics (PPAM 2011), 133–142, 2012.
15. Å. Björck, *Numerical Methods for Least Squares Problems*, SIAM,1996.
16. A. Castaldo and R. Whaley, *Scaling LAPACK panel operations using parallel cache assignment*, Proceedings of the 15th AGM SIGPLAN symposium on principle and practice of parallel programming, 2010, pp. 223–232.
17. J. Demmel, L. Grigori, M. Hoemmen, and J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput. **34** (2012), A206–A239, , also available as EECS Department, University of California, Berkeley, Technical report (UCB/EECS-2008-89).
18. L. Grigori, J. Demmel, and H. Xiang, *CALU: a communication optimal LU factorization algorithm*, SIAM. J. Matrix Anal. Appl. **32** (2011), no. 4, 1317–1350.
19. F. Gustavson, *Recursive leads to automatic variable blocking for dense linear-algebra algorithms*, IBM Journal of Research and Development **41** (1997), 737–755.
20. N. J. Higham, *Accuracy and stability of numerical algorithms*, SIAM, 2002.
21. Nédélec, J.-C., *Acoustic and electromagnetic equations. Integral representations for harmonic problems*, Appl. Math. Sci., vol. 144, Springer-Verlag, New-York, 2001.
22. D. S. Parker, *Random Butterfly Transformations with Applications in Computational Linear Algebra*, Technical Report CSD-950023, UCLA Computer Science Department, 1995.
23. M. Rozložník, G. Shklarski, and S. Toledo, *Partitioned triangular tridiagonalization*, ACM Trans. Math. Softw. **37** (2011), no. 4, 1–16.
24. S. Tomov and J. Dongarra, and M. Baboulin, *Towards dense linear algebra for hybrid GPU accelerated manycore systems*, Parallel Computing, 36(5&6):232–240, 2010.
25. Sivan Toledo, *Locality of reference in LU decomposition with partial pivoting*, SIAM J. Matrix Anal. Appl. **18** (1997), no. 4, 1065–1081.
26. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures*, Version 2.3, 2010. University of Tennessee.