Towards a Parallel Tile LDL Factorization for Multicore Architectures

Dulceneia Becker
University of Tennessee,
Knoxville, TN, USA
dbecker7@eecs.utk.edu

Mathieu Faverge
University of Tennessee,
Knoxville, TN, USA
faverge@eecs.utk.edu

Jack Dongarra
University of Tennessee,
Knoxville, TN, USA
dongarra@eecs.utk.edu

ABSTRACT

The increasing number of cores in modern architectures requires the development of new algorithms as a means to achieving concurrency and hence scalability. This paper presents an algorithm to compute the LDL^T factorization of symmetric indefinite matrices without taking pivoting into consideration. The algorithm, based on the factorizations presented by Buttari *et al.* [11], represents operations as a sequence of small tasks that operate on square blocks of data. These tasks can be scheduled for execution based on dependencies among them and on computational resources available. This allows an out of order execution of tasks that removes the intrinsically sequential nature of the factorization. Numerical and performance results are presented. Numerical results were limited to matrices for which pivoting is not numerically necessary. A performance comparison between LDL^T, Cholesky and LU factorizations and the performance of the kernels required by LDL^T, which are an extension of level-3 BLAS kernels, are presented.

Keywords
LDLT factorization, tile algorithm, BLAS, threaded parallelism, dynamic scheduling

1. INTRODUCTION

The emergence of multicore architectures [28] prompted the need for developing algorithms that lend themselves to parallel execution in shared memory architectures. A class of such algorithms, called *Tile Algorithms*, has been developed for one-sided dense factorizations [12,13] and made available as part of the Parallel Linear Algebra Software for Multicore Architectures (PLASMA) library [29], PLASMA contains Cholesky, LU and QR factorizations. This paper proposes a tile LDL^T algorithm to factorize symmetric matrices such as $A = LDL^T$.

Although symmetric matrices can be decomposed using methods for general matrices, *e.g.* LU and QR, algorithms that exploit symmetry usually require fewer flops and less storage. For instance, the LDL^T factorization presented in [17, Algorithm 4.1.2, p. 139] requires $n^3/3$ flops, about half the number of flops involved in a Gaussian elimination.

An LDL^T factorization can be defined as a variant of the LU factorization in which A is factored into a three-matrix product LDM^T where D is diagonal and L and M are unit lower triangular. If A is symmetric, then $L = M$ and the work associated with the factorization is half of that required by Gaussian elimination [17, p. 135]. If A is symmetric and positive definite (SPD) then D is also positive definite and the factorization can be computed in a stable and efficient manner [16]. For this case a more specialized factorization is available, namely the Cholesky factorization, where $A = LL^T$ and L is lower triangular (not unit).

For symmetric indefinite matrices, D becomes indefinite and the LDL^T factorization may be unstable. Furthermore, a factorization may not even exist for certain nonsingular A, *e.g.* matrices with zeros on the diagonal. To overcome these issues, pivoting strategies can be adopted. There is extensive literature on the subject and several pivoting strategies available [3, 7, 8, 21]. However, for the first version of the tile LDL^T algorithm, no pivoting has been used. This initial development upmostly aims on measuring the performance of tile LDL^T by comparing it to the tile LU and Cholesky algorithms. This can be considered the upper bound in terms of performance, since pivoting improves stability but compromises performance.

Symmetric negative definite matrices (SND) can be decomposed in a stable manner using the LDL^T factorization without pivoting. Since A is definite, D is also definite and hence the factorization is numerically stable. Though the Cholesky factorization can also be used, it has to be applied to $-A$ ($-A$ being SPD), resulting in $A = -(LL^T)$. The LDL^T factorization requires roughly the same number of flops as Cholesky and there is no need to know beforehand if A is SPD or SND.

While many implementations of the LDL^T factorization have been proposed for sparse solvers on distributed and shared memory architectures [15,18,20,26], dense solutions are scarce. Widely known libraries like SCALAPACK, PLASMA and FLAME have implemented solutions for the common Cholesky, LU and QR factorizations [5,14,19,24] but none of them introduced a solution for indefinite symmetric matrices in spite of the gain of flops it could provide for these cases. The main reason to this is the algorithms used for pivoting LDL^T, which are difficult to parallelize efficiently. To our knowledge, the only research in the subject has has been done by Strazdins [27] and the procedure available in the OpenMP version of MKL.

The rest of this paper is organized as follows. In Section 2, the naming convention used throughout this paper is presented. In

Section 3 the LDL^T algorithm is introduced, the tile LDL^T algorithm is detailed and the UDU^T algorithm is also presented, as well as both dynamic and static scheduling. In Section 4, the new level-3 BLAS based kernels required by the LDL^T algorithm are reported. Numerical results regarding a time comparison among LDL^T, Cholesky and LU factorization, the LDL^T scalability, tile size performance and numerical accuracy are presented in Section 5. Conclusion and future work directions are presented in Section 6.

2. NAMING CONVENTION

The naming convention used throughout this paper follows the naming scheme of LAPACK [2]. Names are of the form XYYZZZz, where X indicates the data type, YY the matrix type, ZZZ the computation performed, and z a specialization of the computation performed. Table 1 lists the main codes used for X and YY. ZZZ meaning is explained as used except for the factorization routines that are named:

- xPOTRF: Cholesky factorization

- xSYTRF/xHETRF: LDL^T factorization for symmetric real and hermitian complex matrices, respectively. For clarity of reading, throughout the text only xSYTRF is used although it may refer to xHETRF.

- xGETRF: LU factorization

X	Data Type	YY	Matrix type
S	Real	SY	Symmetric
D	Double Precision	HE	Hermitian
C	Complex	GE	General
Z	Double Complex	TR	Triangular
x	Any data type	PO	Symmetric or Hermitian Positive Definite

Table 1: Data and matrix type naming scheme

3. TILE LDL^T FACTORIZATION

The LDL^T factorization is given by equation

$$A = LDL^T \quad (1)$$

where A is an $N \times N$ symmetric squared matrix, L is unit lower triangular and D is diagonal. For simplicity and also because pivoting is not used, the assumption that D is diagonal has been made, although D can also be block-diagonal, with blocks of size 1×1 or 2×2, when pivoting is applied. The matrix A can also be factorized as $A = UDU^T$, where U is unit upper triangular. The algorithm for the lower triangular case (L) can straightforwardly be extended to the upper triangular case (U) and therefore only the development of the former is presented.

Algorithm 1 shows the steps needed to decompose A using a column-wise LDL^T factorization. After N steps of Algorithm 1, L and D are such

$$A = \begin{bmatrix} 1 & & \\ l_{21} & 1 & \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} d_1 & & \\ & d_2 & \\ & & d_3 \end{bmatrix} \begin{bmatrix} 1 & l_{21} & l_{31} \\ & 1 & l_{32} \\ & & 1 \end{bmatrix}$$

As it is depicted in Figure 1, this process is intrinsically sequential.

Algorithm 1 LDL^T Factorization

1: **for** $j = 1$ to N **do**
2:    **for** $i = 1$ to $j - 1$ **do**
3:       $v_i = A_{j,i}A_{i,i}$
4:    **end for**
5:    $v_j = A_{j,j} - A_{j,1:j-1}v_{1:j-1}$
6:    $A_{j,j} = v_j$
7:    $A_{j+1:N,j} = \left(A_{j+1:N,j} - A_{j+1:N,1:j-1}v_{1:j-1} \right) / v_j$
8: **end for**

$$A^{(4)} = \begin{bmatrix} \circ & & & & & & & \\ & \circ & & & & & & \\ & & \circ & & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \\ \circ & \circ & \circ & \times & & & & \end{bmatrix}$$

Figure 1: Sketch of elements to be calculated ($\times$) and accessed ($\circ$) at the fourth step ($k = 4$) of Algorithm 1.

In order to increase parallelism, the tile algorithm starts by decomposing A in $NT \times NT$ tiles[1] (blocks), such as

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1,NT} \\ A_{21} & A_{22} & \dots & A_{2,NT} \\ \vdots & \vdots & \ddots & \vdots \\ A_{NT,1} & A_{NT,2} & \dots & A_{NT,NT} \end{bmatrix}_{N \times N} \quad (2)$$

Each A_{ij} is a tile of size $MB \times NB$. The same decomposition can be applied to L and D. For instance, for $NT = 3$:

$$L = \begin{bmatrix} L_{11} & & \\ L_{21} & L_{22} & \\ L_{31} & L_{32} & L_{33} \end{bmatrix}, D = \begin{bmatrix} D_{11} & & \\ & D_{22} & \\ & & D_{33} \end{bmatrix}$$

Upon this decomposition and using the same principle of the Schur complement, the following equalities can be obtained:

$$A_{11} = L_{11}D_{11}L_{11}^T \quad (3)$$
$$A_{21} = L_{21}D_{11}L_{11}^T \quad (4)$$
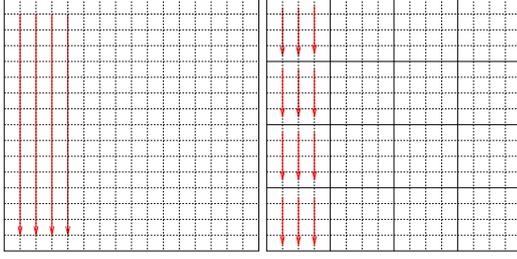$$A_{31} = L_{31}D_{11}L_{11}^T \quad (5)$$
$$A_{22} = L_{21}D_{11}L_{21}^T + L_{22}D_{22}L_{22}^T \quad (6)$$
$$A_{32} = L_{31}D_{11}L_{21}^T + L_{32}D_{22}L_{22}^T \quad (7)$$
$$A_{33} = L_{31}D_{11}L_{31}^T + L_{32}D_{22}L_{32}^T - L_{33}D_{33}L_{33}^T \quad (8)$$

By further rearranging the equalities, a series of tasks can be set to

---

[1]For rectangular matrices A is decomposed into $MT \times NT$ tiles.

Figure 2: LAPACK **and tile layout sketch.**

calculate each L_{ij} and D_{ii}:

$$[L_{11}, D_{11}] = \text{LDL}(A_{11}) \tag{9}$$

$$L_{21} = A_{12}(D_{11}L_{11}^T)^{-1} \tag{10}$$

$$L_{31} = A_{13}(D_{11}L_{11}^T)^{-1} \tag{11}$$

$$\tilde{A}_{22} = A_{22} - L_{21}D_{11}L_{21}^T \tag{12}$$

$$[L_{22}, D_{22}] = \text{LDL}(\tilde{A}_{22}) \tag{13}$$

$$\tilde{A}_{32} = A_{32} - L_{31}D_{11}L_{21}^T \tag{14}$$

$$L_{32} = \tilde{A}_{32}(D_{22}L_{22}^T)^{-1} \tag{15}$$

$$\tilde{A}_{33} = A_{33} - L_{31}D_{11}L_{31}^T + L_{32}D_{22}L_{32}^T \tag{16}$$

$$[L_{33}, D_{33}] = \text{LDL}(\tilde{A}_{33}) \tag{17}$$

where $\text{LDL}(X_{kk})$ at Equations (9), (13) and (17) means the actual LDL^T factorization of tile X_{kk}. These tasks can be executed out of order, as long as dependencies are observed, rendering parallelism.

The decomposition into tiles allows the computation to be performed on small blocks of data that fit into cache. This leads to the need of a reorganization of data formerly given in the LAPACK layout (also known as column major layout) as shown in Figure 2. The tile layout reorders data in such a way that all data of a single block is contiguous in memory. Thus the decomposition of the computation can either be statically scheduled to take advantage of cache locality and reuse or be dynamically scheduled based on dependencies among data and computational resources available. This allows an out of order execution of tasks that removes the intrinsically sequential nature of the factorization of dense matrices.

3.1 Tile LDL^T Algorithm
The tiled algorithm for the LDL^T factorization is based on the following operations:

xSYTRF: This LAPACK based subroutine is used to perform the LDL^T factorization of a symmetric tile A_{kk} of size $NB \times NB$ producing a unit triangular tile L_{kk} and a diagonal tile D_{kk}. Using the notation *input* $\longrightarrow$ *output*, the call xSYTRF(A_{kk}, L_{kk}, D_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, D_{kk} = \text{LDL}^T (A_{kk})$$

xSYTRF2: This subroutine first calls xSYTRF to perform the factorization of A_{kk} and then multiplies L_{kk} by D_{kk}. The call xSYTRF2(A_{kk}, L_{kk}, D_{kk}, W_{kk}) will perform

$$A_{kk} \longrightarrow L_{kk}, D_{kk} = \text{LDL}^T (A_{kk}), \quad W_{kk} = L_{kk}D_{kk}$$

xTRSM: This BLAS subroutine is used to apply the transformation computed by xSYTRF2 to an A_{ik} tile by means of a triangular system solve. The call xTRSM(W_{kk}, A_{ik}) performs

$$W_{kk}, A_{ik} \longrightarrow L_{ik} = A_{ik}W_{kk}^{-T}$$

xSYDRK: This subroutine is used to update the tiles A_{kk} in the trailing submatrix by means of a matrix-matrix multiply. It differs from xGEMDM by taking advantage of the symmetry of A_{kk} and by using only the lower triangular part of A and L. The call xSYDRK(A_{kk}, L_{ki}, D_{ii}) performs

$$A_{kk}, L_{ki}, D_{ii} \longrightarrow A_{kk} \leftarrow A_{kk} - L_{ki}D_{ii}L_{ki}^T$$

xGEMDM: This subroutine is used to update the tiles A_{ij} for $i \neq j$ in the trailing submatrix by means of a matrix-matrix multiply. The call xGEMDM(A_{ij}, L_{ik}, L_{jk}, D_{kk}) performs

$$A_{ij}, L_{ik}, L_{jk}, D_{kk} \longrightarrow A_{ij} \leftarrow A_{ij} - L_{ik}D_{kk}L_{jk}^T$$

Given a symmetric matrix A of size $N \times N$, NT as the number of tiles, such as in Equation (2), and making the assumption that $N = NT \times NB$ (for simplicity), where $NB \times NB$ is the size of each tile A_{ij}, then the tiled LDL^T algorithm can be described as in Algorithm 2. A graphical representation of Algorithm 2 is depicted in Figure 3.

Algorithm 2 Tile LDL^T Factorization

1: **for** $k = 1$ to NT **do**
2:     xSYTRF2(A_{kk}, L_{kk}, D_{kk}, W_{kk})
3:     **for** $i = k+1$ to NT **do**
4:         xTRSM(W_{kk}, A_{ik})
5:     **end for**
6:     **for** $i = k+1$ to NT **do**
7:         xSYDRK(A_{kk}, L_{ki}, D_{ii})
8:         **for** $j = k+1$ to $i-1$ **do**
9:             xGEMDM(A_{ij}, L_{ik}, L_{jk}, D_{kk})
10:         **end for**
11:   **end for**
12: **end for**

3.2 Tile UDU^T Algorithm
The UDU^T algorithm follows the development of the LDL^T algorithm with one major distinction. The LDL^T algorithm starts on tile A_{11}, *i.e.* it proceeds essentially forward (from the top-left to the bottom-right of the matrix) since the first and only task available at the beginning is the factorization of tile A_{11}. For $NT = 2$, the following must be computed:

$$\text{LDL}^T = \left[\begin{array}{c|c} L_{11}D_{11}L_{11}^T & \\ \hline L_{21}D_{11}L_{11}^T & L_{21}D_{11}L_{21}^T + L_{22}D_{22}L_{22}^T \end{array} \right]$$

One can notice that only $L_{11}D_{11}L_{11}^T$ (tile equivalent to A_{11}) can be calculated since L_{11} and D_{11} are required by the other tiles.

For the UDU^T algorithm with $NT = 2$, the following must be computed:

$$\text{UDU}^T = \left[\begin{array}{c|c} U_{11}D_{11}U_{11}^T + U_{12}D_{22}U_{12}^T & U_{12}D_{22}U_{22}^T \\ \hline & U_{22}D_{22}U_{22}^T \end{array} \right]$$
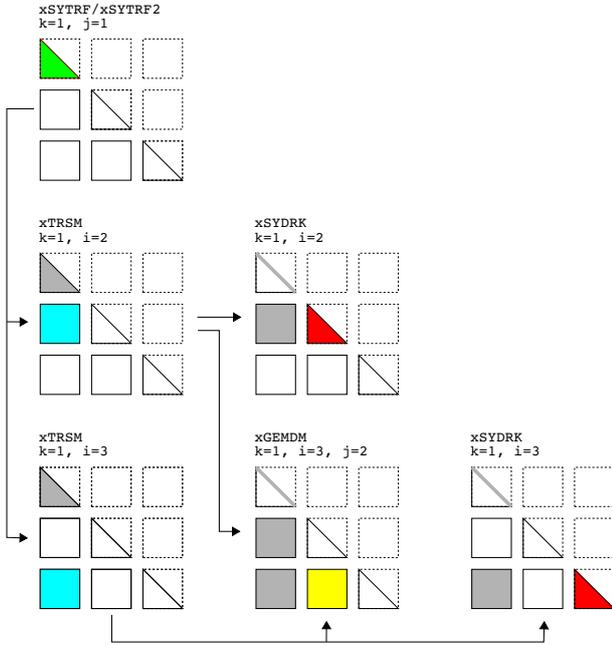
Figure 3: Graphical representation with dependencies of one repetition of the outer loop in Algorithm 2 with $NT = 3$.

Clearly, only $U_{22}D_{22}U_{22}^T$ (tile equivalent to A_{22} or $A_{NT,NT}$ in a more general manner) has no dependencies on the other tiles and hence the computation must proceed backwards (from the bottom-right to the top-left of the matrix). The tile UDU^T algorithm can be described as in Algorithm 3.

Algorithm 3 Tile UDU^T Factorization

1: **for** $k = NT$ to 1 **do**
2:    xSYTRF2($A_{kk}, U_{kk}, D_{kk}, W_{kk}$)
3:    **for** $i = k - 1$ to 0 **do**
4:       xTRSM(W_{kk}, A_{ik})
5:    **end for**
6:    **for** $i = k - 1$ to 0 **do**
7:       xSYDRK(A_{kk}, U_{ki}, D_{ii})
8:       **for** $j = m + 1$ to $k - 1$ **do**
9:          xGEMDM($A_{ij}, U_{ik}, U_{jk}, D_{kk}$)
10:       **end for**
11:    **end for**
12: **end for**


3.3 Static and Dynamic Scheduling

Following the approach presented in [9, 10, 22], Algorithms 2 and 3 can be represented as a Directed Acyclic Graph (DAG) where nodes are elementary tasks that operate on one or several $NB \times NB$ blocks and where edges represent the dependencies among them. A dependency occurs when a task must access data outputted by another task either to update or to read them. Figure 4 shows a DAG for the tile LDL^T factorization when Algorithm 2 is executed with $NT = 3$. Once the DAG is known, the tasks can be scheduled asynchronously and independently as long as the dependencies are not violated.

This dynamic scheduling results in an out of order execution where idle time is almost completely eliminated since only very loose synchronization is required between the threads. Figure 5(a) shows the execution trace of Algorithm 2 where tasks are dynamically scheduled, based on dependencies in the DAG, and run on 8 cores of the *MagnyCours-48* machine (described in Section 5). The tasks were scheduled using QUARK [30], which is the scheduler available in the PLASMA library. Each row in the execution flow shows which tasks are performed and each task is executed by one of the threads involved in the factorization. The trace follows the same color code of Figure 3.

Figure 5(b) shows the trace of Algorithm 2 using static scheduling, which means that each core's workload is predetermined. The synchronization of the computation for correctness is enforced by a global progress table. The static scheduling technique has two important shortcomings. First is the difficulty of development. It requires full understanding of the data dependencies in the algorithm, which is hard to acquire even by an experienced developer. Second is the inability to schedule dynamic algorithms, where the complete task graph is not known beforehand. This is the common situation for eigenvalue algorithms, which are iterative in nature [23]. Finally, it is almost impossible with the static scheduling to overlap simply and efficiently several functionalities like the factorization and the solve that are often called simultaneously. However for a single step, as can be seen on Figure 5, the static scheduling on a small number of cores may outrun the dynamic scheduling due to better data locality and cache reuse.

For the LDL^T factorization, it is possible to build an efficient progress table or execution path. Comparing Figures 5(a) and 5(b) one can notice that the tasks are scheduled in different ways but the execution time is similar (see Section 5 for more details). It is important to highlight that developing an efficient static scheduling can be very difficult and that the dynamic scheduler notably reduces the complexity of programing tile algorithms.
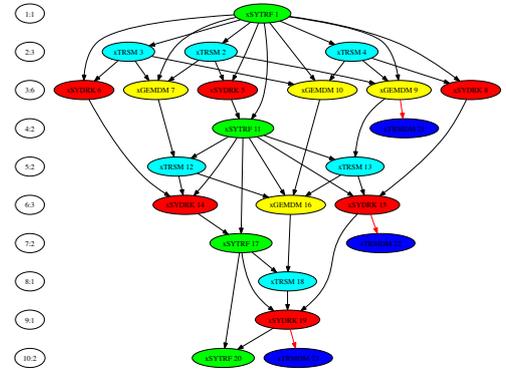

Figure 4: DSYTRF DAG; $NT = 4$.

4. LDL^T KERNELS

The implementation proposed for the LDL^T factorization is similar to the Cholesky procedure, part of the PLASMA library. However, as suggested in Section 3, the LDL^T factorization requires new BLAS-based kernels to deal with the scaling by the diagonal matrix within certain operations. The motivation to develop such kernels is that they are neither available in BLAS libraries nor are part of the extension proposed on the BLAS Technical Forum [4,6]. This extension suggests adding the stand-alone scaling operation
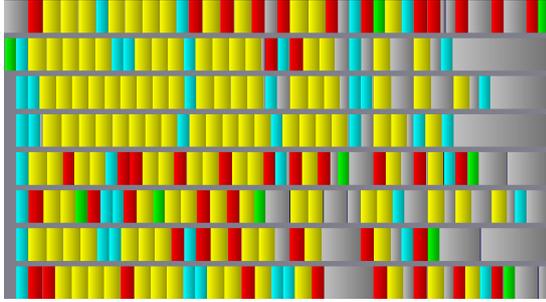
(a) Dynamic scheduling
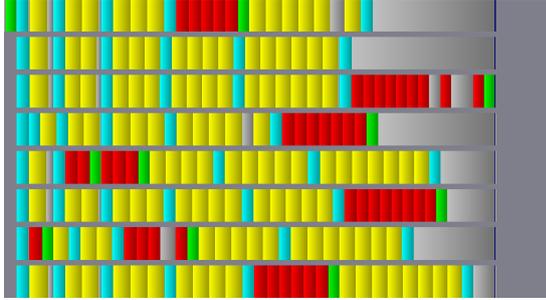
(b) Static scheduling

Figure 5: Traces of DSYTRF (*MagnyCours-48* with 8 threads).

by a diagonal matrix. This does not satisfy the need to include the scaling inside the targeted kernels, namely xGEMM and xSYRK. Furthermore, the extensions still have not been implemented by any vendor libraries to date. The LAPACK library does not require these kernels. Each time LAPACK factorizes a panel L_k, $W_k = L_k D_k$ is temporally stored in a workspace. This workspace is subsequently used during the update of the trailing submatrix to take advantage of the efficiency of level-3 BLAS routines and hence improve performance.

In the case of the tile-algorithm, the same optimization naively applied would require the whole matrix $L \cdot D$ ($N \times N$) to be stored, which represents an extra storage of $N^2/2$. This expensive extra-storage shall be avoided since it can become a considerable limitation; it is known that the non-packed LAPACK storage is already often too expensive for some applications. Thus, new kernels were developed. These kernels require a workspace of size $O(NB^2)$ and, thereby, the memory overhead is limited to $p * NB^2$, where p is the number of cores/threads. This is small compared to the full matrix storage but comes at the cost of extra flops.

Lets define $L_k = (L_{k+1,k} L_{k+2,k} \ldots L_{NT,k})$ the block-column factorization at the step k. In the LAPACK algorithm, the computation of $W_k = L_k D_k$ costs $(NT - k)NB^2$ multiplications and W_k is used several times in the following updates. In the proposed algorithm, the inputs for the update of $A_{ij}, (i, j) \in [k+1, N]^2 \backslash \{i > j\}$ are A_{ij}, L_{ik}, D_k and L_{jk}. Thus, each kernel on row i has to compute its own $L_{ik}D_k$. For the block diagonal update, the same problem occurs except that it requires only half of the flops to update a triangular matrix.

The overhead cost on line i at the step k is therefore

$$row(k, i) = (i - k - 1)NB^2 + \frac{1}{2}NB^2 \qquad (18)$$

which gives the number of multiplications for step k as

$$step(k) = \sum_{i=k+1}^{NT} row(k, i) = \frac{1}{2}NB^2(NT - k)^2 \qquad (19)$$

and hence the total number of extra flops for the entire factorization is

$$\sum_{k=1}^{NT} step(k) \approx \frac{1}{6}NT^3 NB^2 \approx \frac{1}{6 * NB}N^3 \qquad (20)$$

There is a tradeoff between the memory overhead and the extra computation that is heavily dependent on the blocking size NB. An efficient multicore implementation of the LDL^T factorization thus relies on an efficient implementation of the kernels xSYDRK and xGEMDM in order to hinder the additional computation. To achieve this goal, the kernels have been implemented by relying on an efficient xGEMM from the vendor libraries and classic block algorithms used in LAPACK to use level 3 BLAS.

The xGEMDM kernel computes

$$C = \beta C + \alpha \, op_A(A) \cdot D \cdot op_b(B)$$

in two steps. Firstly, either $W = A \cdot D$ or $W = D \cdot op_B(B)$ is computed, for A non-transpose and A transpose respectively. This allows a better scaling by operating on contiguous memory chunks. Once this operation is completed, the level 3 BLAS xGEMM routine is called and performs either the operation

$$C = \beta C + \alpha W \cdot op_b(B)$$

or

$$C = \beta C + \alpha \, op_A(A) \cdot W$$

The xSYDRK kernel calculates either

$$C = \beta C + \alpha A \cdot D \cdot conj(A^T)$$

or

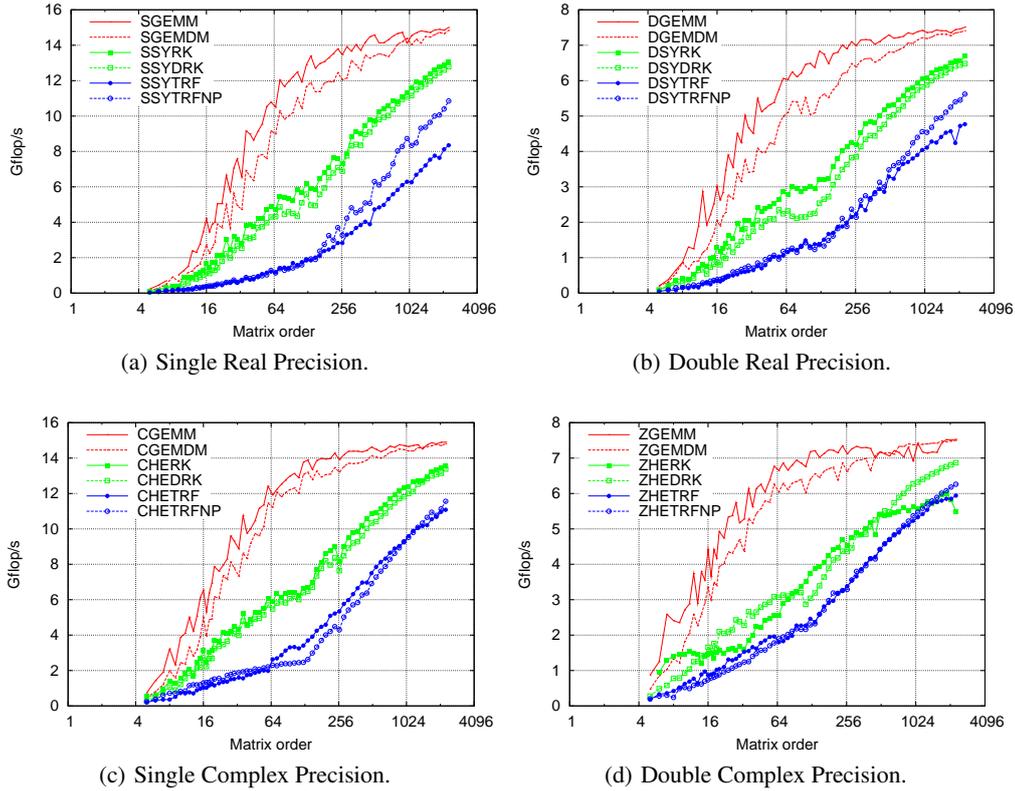$$C = \beta C + \alpha \, conj(A^T) \cdot D \cdot A$$

where C is symmetric and therefore only the lower or upper triangular part is stored. Following the LAPACK standards, only the triangular part stored must be referenced. xSYDRK proceeds in a similar fashion to xGEMDM, additionally exploiting symmetry and operating only the lower or upper triangular part of the matrix.

A kernel for the LDL^T factorization without pivoting has also been implemented and called xSYTRFNP. This kernel performs the operation which takes A as input and returns L and D stored in A as in the LAPACK equivalent routine. The implementation of this kernel follows the implementation of the LAPACK kernel without the Bunch-Kaufman algorithm to search pivots and the swap operations.

Numerical results regarding the performance of these new kernels are presented in Section 5.1.

5. NUMERICAL EXPERIMENTS

The LDL^T algorithm presented in Section 3.1 has been implemented by following the software development guidelines of PLASMA, the Parallel Linear Algebra Software for Multicore Architectures library [1]. The numerical results that follow are presented for both a static and a dynamic scheduler (see Section 3.3) and have

(a) Single Real Precision.

(b) Double Real Precision.

(c) Single Complex Precision.

(d) Double Complex Precision.

Figure 6: Performance comparaison of the LDL^T kernels in the four arithmetic precisions against the equivalent level 3 BLAS used in Cholesky.

been carried out on the system *MagnyCours-48*. This machine has NUMA architecture and is composed of four AMD Opteron 6172 Magny-Cours CPUs running at 2.1GHz with twelve cores each (48 cores total) and 128GB of memory. The theoretical peak of this machine is 403.2 Gflop/s (8.4 Gflop/s per core) in double precision. Comparisons are made against the latest parallel version of the Intel MKL 10.3.2 library released in January 2011 [25], and against the reference LAPACK 3.2 from Netlib, linked against the same MKL BLAS threaded library. PLASMA is linked against the sequential version of MKL for the required BLAS and LAPACK routines. This version of MKL achieves 7.5 Gflop/s on a `DGEMM` (matrix-matrix multiplication) on one core.

Unless otherwise stated, the measurements were carried out using all the 48 cores of the *MagnyCours-48* system and run with `numactl -interleaved=0-#`. Also, a tile size of $NB = 250$ and an inner-blocking size of $IB = 125$.

5.1 Kernels

Figure 6 shows a comparison of the three new kernels implemented for the LDL^T factorization against their *equivalent* without the diagonal matrix to ascertain the overhead introduced by the scaling within the computation. The matrix size ranges from $N = 5$ to $N = 2048$. These relatively small sizes were chosen because these kernels operate only on tiles, which are most commonly less than 2000.

For this set of experiments, the inner blocking parameter *IB*, required by `xSYDRK` and `xSYTRFNP` has been set to 128. This value

has been chosen based on measurements ranging between 32 and 128. Performance was very similar over the entire range, with minor variations, and hence the size of *IB* does not have a significant impact on the performance of kernels.

In order to maximize the performance of `xPOTRF` on the updating, *i.e.* the `xGEMM` step, tile sizes are usually greater than 250 for real and single complex precision, and greater than 120 for double complex precision. The experiments show that for this range of tile sizes, the loss in performance of `xGEMDM` compared to `xGEMM` is less than 1 Gflop/s for single precision and less than half a Gflop/s for double precision.

The performance of `xSYDRK` is very close to the performance of the `xSYRK` kernel from MKL, which means its performance is optimal in comparison to the original kernel. Actually, `xSYDRK` achieves better performance than `xSYRK` for large cases in double complex precision, meaning that the block algorithm used probably has less cache misses than the original kernel.

In summary, as expected, the four kernels for the LDL^T factorization (`xSYTRFNP`) show better performance than the original kernel from MKL, since there is no pivot searching. For comparison purposes, symmetric positive definite matrices have been used to avoid swapping in the MKL kernels and reduce the difference between the two routines. One observes on the results that the pivot search somewhat hinders for the complex precision where the volume of computation is higher. However, it is still visible on the other precisions where the memory latency is more significant.
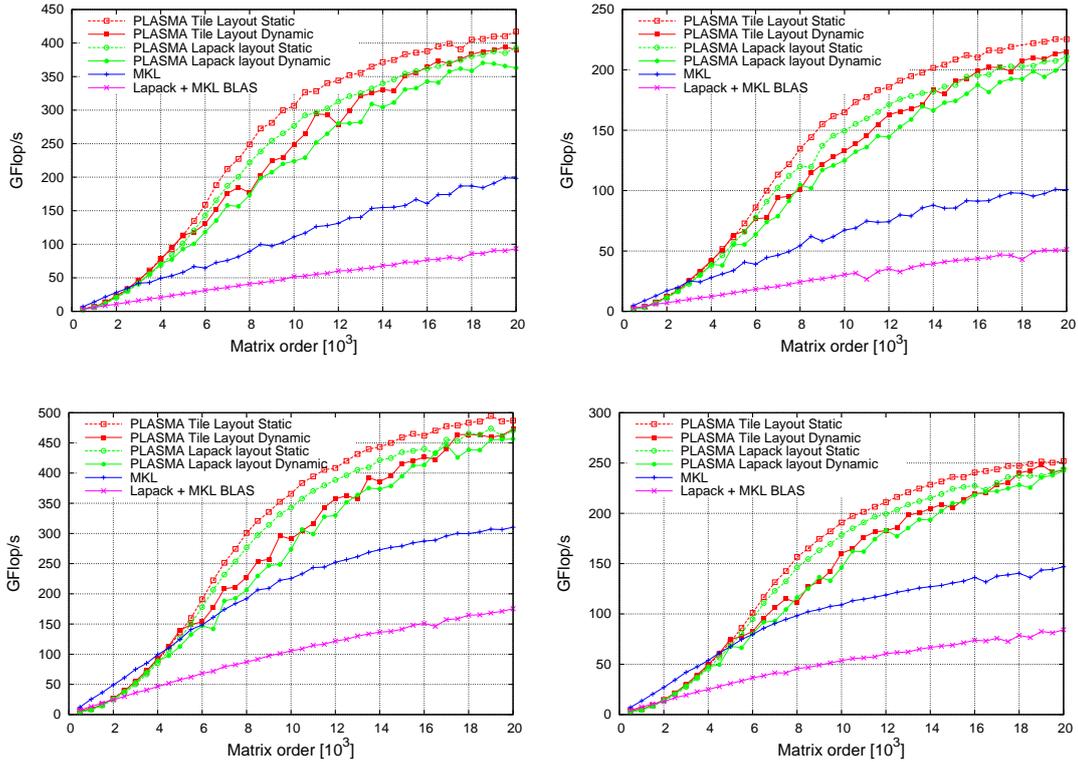
Figure 7: `xSYTRF`/`xHETRF` **performance against MKL and** LAPACK**.**

The three new kernels give good performance and show that in spite of the extra computations, we could expect good efficiency on the overall factorization.

5.2 LDL$^{\mathrm{T}}$ Factorization

Figure 7 shows the performance of PLASMA's tile-LDL$^{\mathrm{T}}$ against both MKL and LAPACK LDL$^{\mathrm{T}}$. The routines are slightly different, since PLASMA's implementation does not include pivoting, while the other two do. However, a definite matrix has been chosen for performance comparison so that no permutations are actually made by all the three implementations, *i.e.* MKL and LAPACK perform searches but do not do permutations. Four cases of the PLASMA implementation are reported: LAPACK and tile layout, each with dynamic and static scheduling. For the LAPACK layout, data is created and initially stored contiguously column-wise in a single block $N \times N$ and then translate into tile layout and back to LAPACK layout after computation is completed. For the tile layout, data is stored in block $NB \times NB$ (tiles) from the beginning and no translation is needed. The factorization itself is done using the tile layout in both cases.

The overall performance for the four PLASMA cases is quite similar. The static scheduling usually outruns the dynamic one, mostly due to the overhead of the dynamic scheduler. As mentioned before, the progress table for tile LDL$^{\mathrm{T}}$ is quite efficient, exposing the overhead caused by the dynamic scheduler. The LAPACK layout includes the translation step and is therefore expected to be outrun by the tile layout. Nevertheless, the difference is almost negligible. Regarding MKL and LAPACK, PLASMA is about twice as fast compared against MKL and four times as fast as LAPACK for all

the four precisions presented in Figure 7.

Figure 8 reports the execution time of `xSYTRF`, `xPOTRF` and `xGETRF` with dynamic and static scheduling. The static scheduling scheme usually delivers the highest performance. This happens mostly because there is no overhead on scheduling the tasks and, as mentioned before, the LDL$^{\mathrm{T}}$ algorithm lends itself a quite efficient progress table. As expected, LDL$^{\mathrm{T}}$ is noticeably faster than LU and only moderately slower than Cholesky. This clearly states that it is advantageous, in terms of time, to factorize a symmetric matrix using `xSYTRF` (instead of `xGETRF`) and also that `xSYTRF` (instead of `xPOTRF`) can be used on decomposing SPD matrices with no substantial time overhead.

The parallel speedup or scalability of `xSYTRF` is shown in Figure 9 for matrices of order 5000, 10000 and 20000 [N], both for dynamic and static scheduling. As anticipated, the parallel speedup increases as the matrix order increases. This happens because the bigger the matrix, the more tasks are available to be executed concurrently, resulting in higher scalability. The parallel performance actually depends on several factors, one of them being the tile size [NB]. As seen in Figure 6, the individual kernerls achieve better performance as the size of the matrix (or tiles) increases, suggesting that bigger tiles should be used. However, as the size of the tiles increases, the number of tasks to be executed concurrently decreases, which means less granularity. A trade-off between the performance of individual kernels and the overall granularity must be made.

The performance reported has been obtained with $NB = 250$ and $IB = 125$. As depicted in Figure 10, 250 is not necessarily the
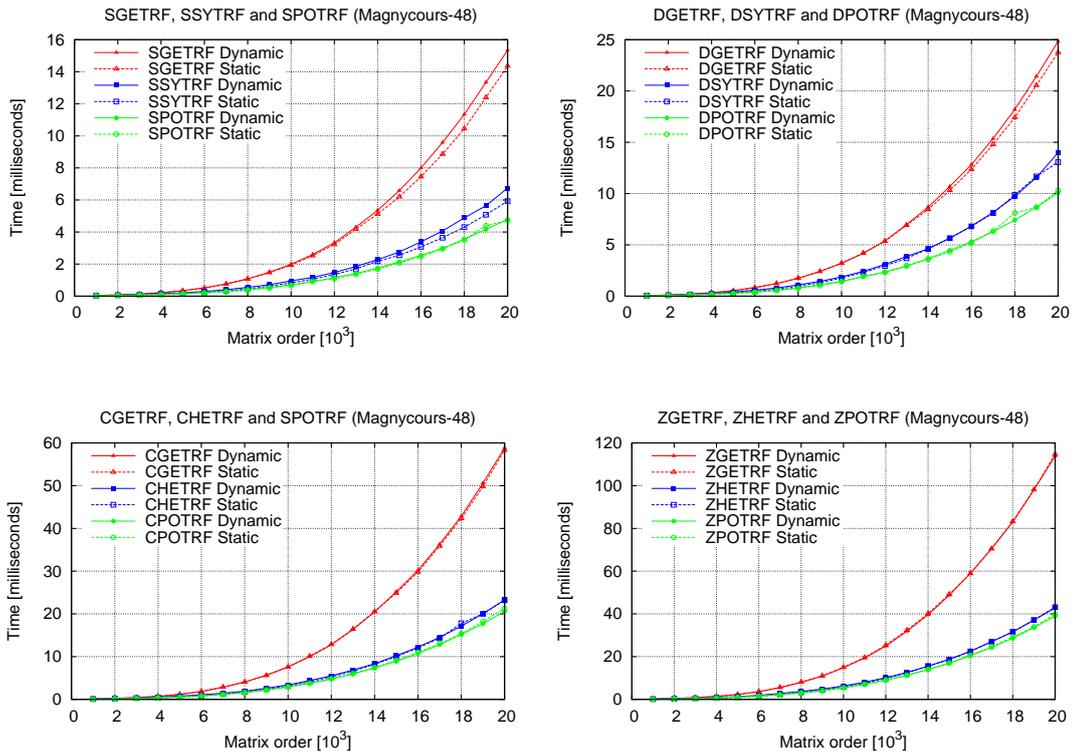
Figure 8: Execution time of `xPOTRF`, `xSYTRF`/`xHETRF` **and** `xGETRF`; **dynamic (solid line) and static (dashed line) scheduling.**
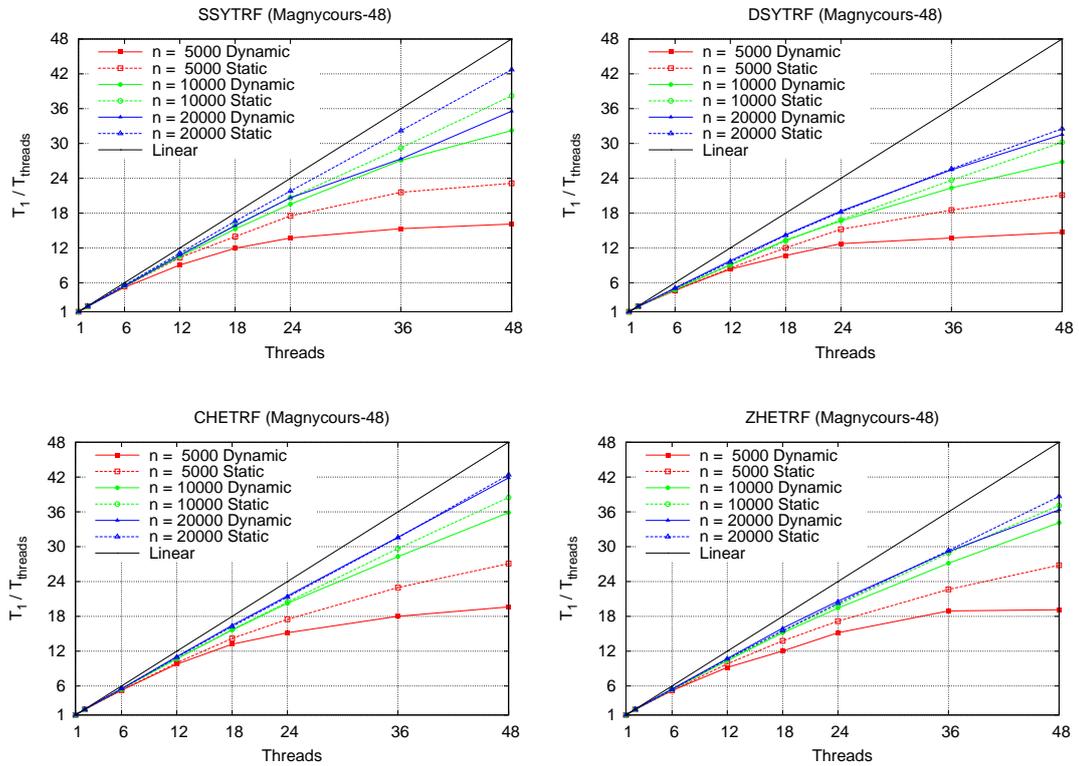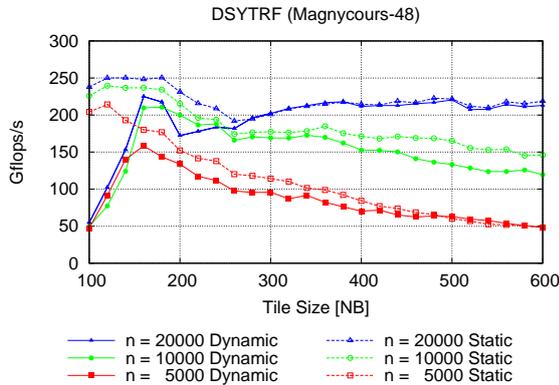

Figure 9: `xSYTRF`/`xHETRF` **scalability; dynamic (solid line) and static (dashed line) scheduling.**

optimal tile size[2]. In order to achieve optimal performance, *NB* and other parameters must be tuned accordingly to the size of the matrix to be decomposed and the number of threads. This could be achieved through auto-tuning; this feature is still not available however.

Figure 10: Tile-size performance of DSYTRF, dynamic (solid line) and static (dashed line) scheduling ($IB = 125$).

The numerical accuracy of a tile algorithm depends strongly on the matrix being factorized and the number of tiles used [*NT*]. In general, the partitioning of the matrix in tiles implies reduced accuracy and the more tiles the less accurate the solution. For the LDL^T algorithm presented in this work, there are also issues with numerical instability since no other technique has been used to overcome such instabilities. About 30% of the test cases in LAPACK failed to achieve a given threshold. Most of the cases that failed had zeros on the diagonal of the matrix, which means that the factorization cannot be completed. This is not the subject of this work however. Table 2 shows the residual of a system where *A* is symmetric positive definite (SPD), the right hand side *b* has been chosen arbitrarily and *x* has been calculated upon the factorization of *A* by xSYTRF or xPOTRF. Matrix *A* has been chosen SPD to measure only the penalty due to partitioning the matrix and executing the task in an out of order manner, without numerical instability. For this experiment, $NB = 1000$ and $IB = 100$. The first row of Table 2 [$NT = 1$] mimics the factorization applied to the entire matrix (without tiles). It can be seen that, though the residual increases as *NT* increases, the loss of accuracy is quite low and it is actually less than for xPOTRF.

6. CONCLUSION AND FUTURE WORK

A tile LDL^T algorithm and its implementation in the PLASMA library were presented. Although the current algorithm is still unstable for a number of matrices, it has been shown that performance-wise it is scalable and achieves almost the same performance of the tile Cholesky algorithm. Since Cholesky can be taken as an upper bound for LDL^T, this shows that the algorithm, together with its implementation, has achieved satisfactory results. It has also been shown that the LDL^T factorization is considerably faster than the LU factorization. Although the algorithm presented does not include pivoting or any other strategy to increase numerical stability, it is unlikely that the penalty for adding such methods will slow the algorithm down by a factor of two or more.

---

[2]Similar results were obtained for single real, single complex and double complex precision.

	||Ax − b||$_2$			
NT	SSYTRF	SPOTRF	DSYTRF	DPOTRF
1	1.71×10^{-6}	2.00×10^{-6}	3.62×10^{-15}	3.26×10^{-15}
2	3.26×10^{-6}	3.24×10^{-6}	5.48×10^{-15}	5.77×10^{-15}
3	5.35×10^{-6}	4.62×10^{-6}	7.72×10^{-15}	8.08×10^{-15}
4	5.54×10^{-6}	4.28×10^{-6}	7.72×10^{-15}	8.56×10^{-15}
5	4.65×10^{-6}	4.94×10^{-6}	9.41×10^{-15}	7.82×10^{-15}
6	5.84×10^{-6}	6.11×10^{-6}	1.14×10^{-14}	1.05×10^{-14}
7	6.73×10^{-6}	6.11×10^{-6}	1.66×10^{-14}	1.10×10^{-14}
8	6.85×10^{-6}	6.62×10^{-6}	1.52×10^{-14}	1.27×10^{-14}
9	7.43×10^{-6}	9.06×10^{-6}	1.34×10^{-14}	1.40×10^{-14}
10	7.44×10^{-6}	9.63×10^{-6}	1.47×10^{-14}	1.70×10^{-14}

Table 2: Residual of symmetric positive definite system, single and double real precision LDLT (xSYTRF) and Cholesky (xPOTRF); $N = NT \times 10^3$.

The partitioning of the matrix in tiles in order to increase parallelism does slightly reduce accuracy. However, the partitioning in tiles and consequent out of order execution does not compromise the solution. Also, for a range of matrices numerical stability is not an issue, *e.g.* both positive- and negative-definite matrices. The kernels introduced perform well, but improvements are under consideration. As future work, the use of pivoting techniques or some other way to make the algorithm numerically stable and hence increase the range of matrices that can be solved shall be investigated. Pivoting techniques are the most common choice. However, as mentioned before, these methods can be extremely difficult to parallelize. Therefore, other alternatives shall be considered.

7. REFERENCES

[1] PLASMA. http://icl.cs.utk.edu/plasma/.
[2] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, 1992. http://www.netlib.org/lapack/lug/.
[3] C. Ashcraft, R. G. Grimes, and J. G. Lewis. Accurate symmetric indefinite linear equation solvers. *SIAM J. Matrix Anal. Appl.*, 20:513–561, April 1999.
[4] Basic Linear Algebra Technical Forum. *Basic Linear Algebra Technical Forum Standard*, August 2001. http://www.netlib.org/blas/blast-forum/blas-report.pdf.
[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, PA, 1997. http://www.netlib.org/scalapack/slug/.
[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28:135–151, June 2002.
[7] J. R. Bunch and L. Kaufman. Some stable methods for calculating inertia and solving symmetric linear systems. *Mathematics of Computation*, 31(137):pp. 163–179, 1977.
[8] J. R. Bunch and B. N. Parlett. Direct methods for solving symmetric indefinite systems of linear equations. *Siam*

Journal on Numerical Analysis, 8, 1971.

[9] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. *PARA 2006*, 2006.

[10] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, Vol. 20:pp. 1573–1590, 2007.

[11] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 191, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, Sept. 2007.

[12] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency Computat.: Pract. Exper.*, 20(13):1573–1590, 2008. DOI: 10.1002/cpe.1301.

[13] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parellel Comput. Syst. Appl.*, 35:38–53, 2009. DOI: 10.1016/j.parco.2008.10.002.

[14] J. Choi, J. J. Dongarra, S. Ostrouchov, A. Petitet, D. W. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[15] J. W. Demmel, J. R. Gilbert, and X. S. Li. An asynchronous parallel supernodal algorithm for sparse gaussian elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[16] R. Fletcher. Factorizing symmetric indefinite matrices. *Linear Algebra and its Applications*, 14(3):257 – 272, 1976.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, MD, third edition, 1996.

[18] N. I. M. Gould, J. A. Scott, and Y. Hu. A numerical evaluation of sparse direct solvers for the solution of large sparse symmetric linear systems of equations. *ACM Trans. Math. Softw.*, 33, June 2007.

[19] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, Dec. 2001.

[20] P. Hénon, P. Ramet, and J. Roman. On using an hybrid MPI-Thread programming for the implementation of a parallel sparse direct solver on a network of SMP nodes. In *PPAM'05*, volume 3911 of *LNCS*, pages 1050–1057, Poznan, Pologne, Sept. 2005.

[21] N. J. Higham. Stability of the diagonal pivoting method with partial pivoting. *SIAM J. Matrix Anal. Appl.*, 18:52–65, January 1997.

[22] J. Kurzak and J. Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. LAPACK Working Note 178, Department of Computer Science, University of Tennessee, Knoxville, Knoxville, TN 37996, USA, Sept. 2006. Also available as UT-CS-06-581.

[23] J. Kurzak and J. Dongarra. Fully dynamic scheduler for numerical computing on multicore processors. Technical report, Innovative Computing Laboratory, University of Tennessee, 2009.

[24] J. Kurzak, H. Ltaief, J. Dongarra, and R. Badia. Scheduling linear algebra operations on multicore processors. *Concurrency Practice and Experience (to appear)*, 2009.

[25] Intel, Math Kernel Library (MKL). http://www.intel.com/software/products/mkl/.

[26] O. Schenk and K. Gärtner. On fast factorization pivoting methods for symmetric indefinite systems. *Elec. Trans. Numer. Anal.*, 23:158–179475–487, 2006.

[27] P. E. Strazdins. A dense complex symmetric indefinite solver for the fujitsu ap3000. Technical report, 1999.

[28] H. Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), 2005.

[29] University of Tennessee. *PLASMA Users' Guide, Parallel Linear Algebra Software for Multicore Architectures, Version 2.3*, November 2010.

[30] A. YarKhan, J. Kurzak, and J. Dongarra. Quark users' guide: Queueing and runtime for kernels. Technical report, Innovative Computing Laboratory, University of Tennessee, 2011.