A Run-Time System for Programming Out-of-Core Matrix Algorithms-by-Tiles on Multithreaded Architectures

FLAME Working Note #43

Gregorio Quintana-Ortí* Francisco Igual* Mercedes Marqués*
Enrique S. Quintana-Ortí* Robert van de Geijn†

Abstract

Out-of-core implementations of algorithms for dense matrix computations have traditionally focused on optimal use of memory so as to minimize I/O, often trading programmability for performance. In this paper we show how the current state of hardware and software allows the programmability problem to be solved without sacrificing performance. This comes from the realizations that memory is cheap and large, making it less necessary to optimally orchestrate I/O, and that new algorithms view matrices are collections of submatrices and computation as operations with those submatrices. This enables libraries to be coded at a high level of abstraction, leaving the tasks of scheduling the computations and data movement in the hands of a run-time system. Remarkable performance is demonstrated for this approach on multi-core architectures as well as platforms equipped with hardware accelerators.

1 Introduction

While there may not be many applications that give rise to huge dense matrix computations, there is a surprising number that fall into the category of *large*, i.e. with problem size $\sim 100,000$. Examples of these include integral equation-based electromagnetics problems [27], dielectric polarization in nanostructures [21], estimation of Earth's gravitational field [1, 13], boundary element formulations in acoustics [10], and molecular dynamics simulations [25]. Such problems are now small enough that they can be solved, with some patience, on a multithreaded architecture, including multi-processor, multi-core and GPU-accelerated computers. On these architectures the key is as much the time to compute as the memory required to store the data structures. The solution is to store the matrix on disk, requiring so-called *out-of-core* (OOC) computation. Unfortunately, relying on virtual memory, in which the operating system transfers pages between disk and main memory, often yields low performance for this type of operations.

Background The idea of OOC computation for linear solvers dates back to the early days of scientific computing, when the size of main memory of a processor consisted of magnetic rings and was called core memory [4, 24]. Its size was measured in Kbytes so that data was primarily stored on tape.

Traditional OOC algorithms for the LU and QR factorizations typically employ partial row pivoting (for stability) and Householder transformations, respectively [11]. They view the matrix as a collection of slabs (or panels) of columns, each of which fills a substantial fraction of the main memory. Left-looking algorithms feature the property that, at a given point in the computation, all slabs to the left of the "current" slab have been completely processed while all slabs to the right have not yet been accessed. The current slab is then retrieved into memory, updated with respect to previous slabs (which are retrieved into memory by slabs or

*Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain.
†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

by portions of slabs), and finally written out to disk. The key is to fill most of memory with the current slab since this best amortizes the cost of I/O over useful computation. Left-looking OOC algorithms are preferred over right-looking ones because they exhibit a smaller number of writes to disk.

Strictly speaking, slab solvers are not scalable in the sense that as the problem size increases, performance cannot be maintained. Consider a matrix so large that only a single column can fit in main memory. The ratio of I/O to computation then becomes $O(1)$ and the overhead turns out to be prohibitive. An alternative is incorporated in SOLAR [26] which uses recursive algorithms with matrices that are stored by tiles to yield an almost scalable algorithm. Still, a few tiles fill most of memory and I/O needs to be carefully and explicitly orchestrated by the implementation.

There is some theoretical evidence that algorithms-by-tiles incur less I/O overhead than recursive algorithms [5]. OOC algorithms that compute with tiles but use LU with incremental pivoting and incremental QR factorization were already proposed in [14, 19] for distributed-memory architectures. However, there again the size of the tiles was taken so that a few of them filled most of available memory and the I/O was explicitly embedded into the code. With the advent of cheap, very large, memories, this obsession for optimal use of in-core memory is no longer necessary.

Algorithm-by-blocks for multithreaded architectures More recently, our algorithms based on LU with incremental pivoting and incremental QR factorization have targeted multithreaded architectures (see, e.g., [23] and the references therein). We call these algorithms-by-blocks, to capture the idea that matrices are viewed as composed of a collection of blocks (submatrices). There are two differences between algorithms-by-blocks and the tiled OOC algorithms used by, e.g., SOLAR. Specifically, in algorithms-by-blocks, the block sizes are considerably smaller and a separation of concerns is achieved by having an initial execution of the algorithm produce a directed acyclic graph (DAG) of tasks that a run-time system subsequently schedules for execution to idle cores. As part of our FLAME project this has yielded the SuperMatrix run-time system [9, 23]. A similar approach has been adopted by the PLASMA project [7].

This paper The present work differs from traditional OOC algorithms in that it adopts part of the approach of SuperMatrix: It deals with tiles that are considerably smaller than the available memory, it uses the same API (FLASH) and implementations of algorithms-by-blocks, and it is complemented with a run-time system that generates a list of tasks. It differs in a few important ways: First, tiles are much larger than blocks and the data is assumed to start and end on disk. Second, we do not use the run-time system to track dependencies among tasks and schedule for execution those with all dependencies fulfilled (possibly out-of-order). Instead, the run-time system now performs the role of a software memory controller in that it manages the main memory as a software cache, and prefetches data while executing tasks in-order. As a result, the burden of overlapping I/O with computation and/or explicitly developing a code that optimally amortizes I/O is no longer on the programmer's shoulders. In a nutshell: as long as tiles can be prefetched while computation occurs, near optimal performance will result.

The paper is structured as follows. Section 2 reviews the FLAME approach to expressing algorithms and encoding subroutines for dense linear algebra operations. The material there is very similar to that in [14, 23] and is included in this paper to make it self-contained; thus, readers who already familiar with the FLAME project and the SuperMatrix run-time system may want to skip this section. In Section 3 we briefly connect the algorithms-by-blocks for the Cholesky, LU and QR factorizations, presented in the previous section, with the traditional OOC implementations. Section 4 describes the operation of the run-time that manages the main memory as a software cache of data on disk, prefetching tiles from it, and overlapping computation and I/O. In Section 5 we demonstrate the practical performance of the OOC algorithms for the three major factorizations involved in the solution of linear systems on current general-purpose multi-core processors and GPU-accelerated computers. Finally, Section 6 collects the conclusions of this work as well as some potential future lines of investigation.

2 Algorithms for Dense Linear Algebra Factorizations

In this section we describe the algorithms that underlie our OOC implementations.

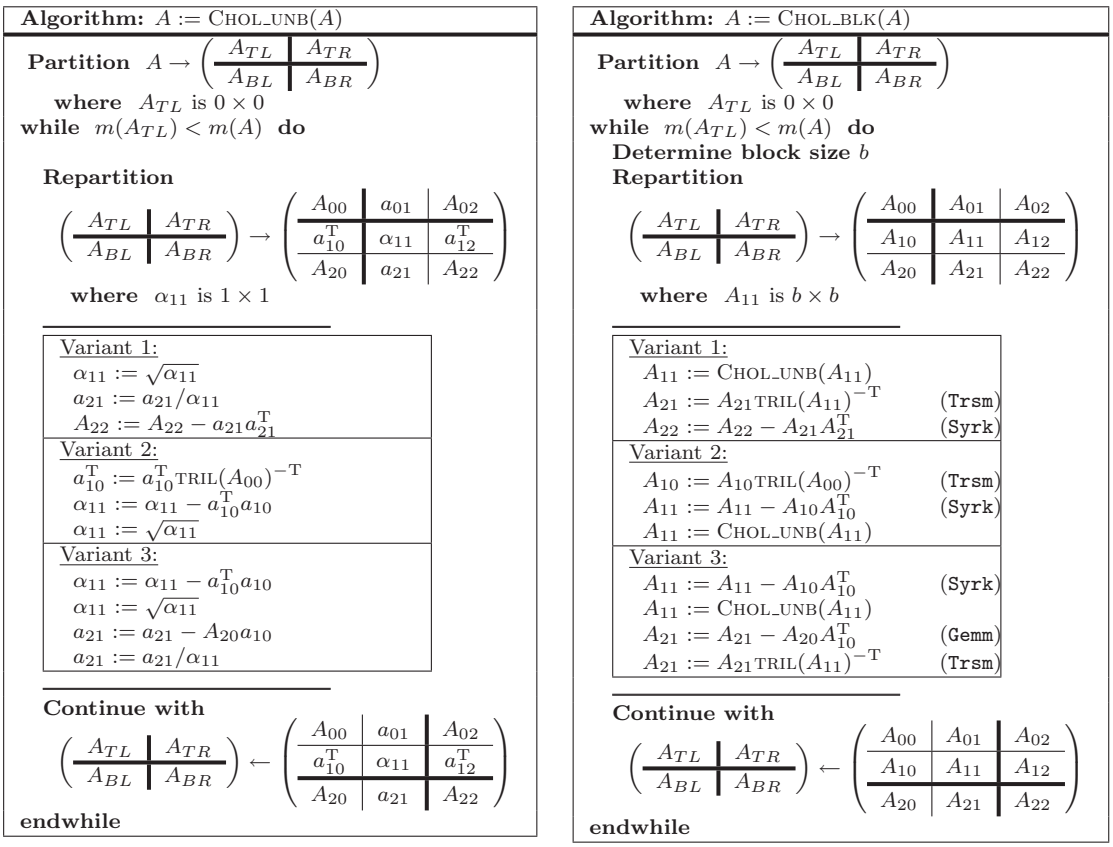


Figure 1: Multiple unblocked (left) and blocked (right) algorithms for computing the Cholesky factorization. In the notation, $m(\cdot)$ stands for the number of rows of a matrix, and TRIL(A) denotes the matrix consisting of the elements in the lower triangular part of A.

2.1 Blocked algorithms vs. algorithms-by-tiles: the Cholesky factorization

High performance algorithms for dense linear algebra operations cast the bulk of the computation in terms of matrix-matrix products in order to exploit the multi-layered structure of the memory system by reusing data that are closer to the processor. In particular, Figure 1 (right) shows the three *blocked algorithms* (variants) for the Cholesky factorization of a symmetric positive definite matrix, specified using the FLAME notation; see, e.g., [12, 6]. All three blocked algorithms in the figure are composed of a main loop which iterates over the contents of the matrix. The variants only differ in some of the updates performed in the body of the loop and the ordering of these operations; they all require the factorization of the Cholesky factorization of the diagonal block A_{11}, which can be computed using any of the three unblocked variants in Figure 1 (left). The other operations appearing in the algorithms are symmetric rank-b updates (`Syrk`), triangular system solves (`Trsm`) and general matrix-matrix products (`Gemm`). Upon completion, the lower triangular part of A is overwritten with the Cholesky factor L such that $A = LL^T$.

Algorithms-by-blocks for dense linear algebra operations also aim at improving data reuse, but from a different perspective: When moving from algorithms that cast most computation in terms of matrix-vector operations to algorithms that mainly operate in terms of matrix-matrix computations, rather than improving performance by aggregating the computation into matrix-matrix computations, the granularity of the data is raised by replacing each element in the matrix by a submatrix (block). Algorithms are then written as before, except with scalar operations replaced by operations on the blocks. In effect, in algorithms-by-blocks, the block becomes the unit of operation.

Algorithms-by-tiles, introduced in this paper, extend this concept by defining the *tile* (a square-like large block) as the unit of transference between (main) memory and disk. Thus, we will make a hierarchical usage of these two concepts: A matrix will be composed of tiles of dimension $t \times t$, with each tile consisting of several blocks of size $b \times b$ ($b < t$). Tiles will be moved from/to disk by an algorithm-by-tiles. Computations will be performed on the blocks that compose an in-core tile (i.e., a tile that resides in memory) by an algorithm-by-blocks.

To obtain an algorithm-by-blocks/tiles for the Cholesky factorization, consider the following partitioning of matrix A:

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{\mathrm{T}} & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|cc} \bar{\alpha}_{00} & \cdots & \bar{\alpha}_{0k} & \cdots & \bar{\alpha}_{0,n-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline \bar{\alpha}_{k0} & \cdots & \bar{\alpha}_{kk} & \cdots & \bar{\alpha}_{k,n-1} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \bar{\alpha}_{n-1,0} & \cdots & \bar{\alpha}_{n-1,k} & \cdots & \bar{\alpha}_{n-1,n-1} \end{array}\right)$$

where α_{11}, $\bar{\alpha}_{ij}$, $0 \leq i, j < n$, are all scalars. The unblocked algorithmic Variant 3 for the Cholesky factorization in Figure 1 (left) can be turned into an algorithm-by-blocks/tiles by recognizing that, if each element in the matrix is itself a matrix, as in

$$A \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right) = \left(\begin{array}{ccc|c|cc} \bar{A}_{00} & \cdots & \bar{A}_{0K} & \cdots & \bar{A}_{0,N-1} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \hline \bar{A}_{K0} & \cdots & \bar{A}_{KK} & \cdots & \bar{A}_{K,N-1} \\ \hline \vdots & \ddots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \cdots & \bar{A}_{N-1,K} & \cdots & \bar{A}_{N-1,N-1} \end{array}\right),$$

where A_{11} and $\bar{A}_{ij}$, $0 \leq i, j < N$, are all $b \times b$ blocks/$t \times t$ tiles; then:

1. α_{11} represents a single block/tile and a_{10}^{T} a row vector of blocks/tiles so that $\alpha_{11} := \alpha_{11} - a_{10}^{\mathrm{T}} a_{10}$ becomes a collection of matrix-matrix products that update A_{11}, one for each block/tile of A_{10}:

$$\begin{array}{rcl} A_{11} & := & A_{11} - A_{10}A_{10}^{\mathrm{T}} = A_{11} - \left(\begin{array}{ccc} \bar{A}_{K0} & \cdots & \bar{A}_{K,K-1} \end{array}\right) \left(\begin{array}{c} \bar{A}_{K0}^{\mathrm{T}} \\ \vdots \\ \bar{A}_{K,K-1}^{\mathrm{T}} \end{array}\right) \\ & = & A_{11} - \bar{A}_{K0}\bar{A}_{K0}^{\mathrm{T}} - \ldots - \bar{A}_{K,K-1}\bar{A}_{K,K-1}^{\mathrm{T}}. \end{array}$$

2. $\alpha_{11} := \sqrt{\alpha_{11}}$ becomes the Cholesky factorization of the matrix block/tile A_{11}:

$$A_{11} := \textsc{Chol}(A_{11}).$$

3. Each element in a_{21} describes a block/tile that is updated with the product of the blocks/tiles in a_{20} and a_{10}^{T}:

$$\begin{array}{rcl} A_{21} & := & A_{21} - A_{20}A_{10}^{\mathrm{T}} \\ & = & \left(\begin{array}{c} \bar{A}_{K+1,K} \\ \vdots \\ \bar{A}_{N-1,K} \end{array}\right) - \left(\begin{array}{ccc} \bar{A}_{K+1,0} & \cdots & \bar{A}_{K+1,K-1} \\ \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \cdots & \bar{A}_{N-1,K-1} \end{array}\right) \left(\begin{array}{c} \bar{A}_{K0}^{\mathrm{T}} \\ \vdots \\ \bar{A}_{K,K-1}^{\mathrm{T}} \end{array}\right). \end{array}$$

4. $a_{21} := a_{21}/\alpha_{11}$ becomes a triangular solve with multiple independent terms, with the updated lower triangular matrix in element α_{11} and each of the blocks/tiles in vector a_{21}:

$$\begin{array}{rcl} A_{21} & := & A_{21}\textsc{tril}(A_{11})^{-\mathrm{T}} \\ & = & \left(\begin{array}{c} \bar{A}_{K+1,K} \\ \vdots \\ \bar{A}_{N-1,K} \end{array}\right) \textsc{tril}(\bar{A}_{KK})^{-\mathrm{T}} = \left(\begin{array}{c} \bar{A}_{K+1,K}\textsc{tril}(\bar{A}_{KK})^{-\mathrm{T}} \\ \vdots \\ \bar{A}_{N-1,K}\textsc{tril}(\bar{A}_{KK})^{-\mathrm{T}} \end{array}\right). \end{array}$$

```
FLA_Error FLASH_Chol_by_tiles_var3( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;


  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {

    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*----------------------------------------------*/
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A10,
                FLA_ONE,       A11 );
    FLA_Chol_blk_var1( FLASH_MATRIX_AT( A11 ), nb_alg );
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                FLA_MINUS_ONE, A20, A10,
                FLA_ONE,       A21 );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    /*----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
FLA_Error FLA_Chol_blk_var1( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       b, b, FLA_BR );
    /*----------------------------------------------*/
    FLA_Chol_unb_var1( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11,
                       A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21,
              FLA_ONE,       A22 );
    /*----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figure 2: FLASH implementation of the algorithm-by-tiles for Variant 3 of the Cholesky factorization (left) and FLAME/C implementation of the blocked algorithm for Variant 1 of the Cholesky factorization (right).

2.2 Transforming algorithms into code with the FLAME APIs

FLAME includes a variety of application programming interfaces (APIs) that allow an easy transition from algorithm to code, reducing the possibility of introducing errors during this process.

A particular instance of the FLAME APIs, FLASH [20], allows the construction of hierarchical matrices, with the entries in these matrices being either scalars or matrices. Using this API, we can accommodate multiple levels in the hierarchy/structure, which will become particularly useful when coding OOC algorithms. An implementation of the algorithm-by-tiles for Variant 3 of the Cholesky factorization using FLASH is offered in Figures 2 and 3. Routine `FLASH_Chol_by_tiles_var3` builds upon the FLAME/C blocked factorization kernel `FLA_Chol_blk_var1` as well as routines `FLASH_Syrk`, `FLASH_Gemm`, `FLASH_Trsm` and `FLASH_Gepp`. Note that in FLASH, A can be built/viewed as a matrix of tiles, with one level in the hierarchy in this case, so that the partitionings operate in terms of tiles, and all computations operate with tiles by invoking kernels `FLA_Syrk` (symmetric rank-t update), `FLA_Trsm` (triangular system solve), and `FLA_Gemm` (general matrix-matrix product). These can be simple wrappers to tuned codes from high performance libraries as, e.g., Intel MKL or GotoBLAS; alternatively, we could also build each tile as a collection of blocks and use algorithms-by-blocks for the implementations of kernels `FLA_Syrk`, `FLA_Trsm` and `FLA_Gemm`, thus leading to a two-level hierarchy.

2.3 Algorithms-by-tiles for the LU and QR factorizations

Unblocked in-core algorithms for the LU and QR factorizations of a matrix A employ, respectively, Gaussian and Householder transforms to annihilate the subdiagonal elements of the matrix, processing one column per iteration, and effectively reducing A to upper triangular form. Left-looking blocked algorithms for these factorizations build upon these procedures to improve data locality: at each iteration, the current panel (or slab) of columns is updated with respect to the previous transforms and then factored.

These two factorizations are not easily formulated in terms of algorithms-by-tiles. The problem arises from the use of partial pivoting in the LU and the computation of Householder transforms for the QR factorization which require access to columns that span multiple tiles. These difficulties were overcome

```
void FLASH_Syrk_ln( FLA_Obj alpha, FLA_Obj A,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                 ...                                   )
   Assumption: A is a row of blocks (row panel) */
{
  FLA_Obj AL,    AR,       A0,  A1,  A2;

  FLA_Part_1x2( A,    &AL,  &AR,      0, FLA_LEFT );

  while ( FLA_Obj_width( AL ) < FLA_Obj_width( A ) ){

    FLA_Repart_1x2_to_1x3(
       AL,  /**/ AR,        &A0, /**/ &A1, &A2,
       1, FLA_RIGHT );
    /*-----------------------------------------------*/
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              alpha, FLASH_MATRIX_AT( A1 ),
              beta,  FLASH_MATRIX_AT( C ) );
    /*-----------------------------------------------*/

    FLA_Cont_with_1x3_to_1x2(
       &AL,  /**/ &AR,        A0, A1, /**/ A2,
       FLA_LEFT );

  }
}
```

```
void FLASH_Trsm_rltn( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                                  )
   Assumption: L consists of one block and
               B consists of a column of blocks     */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;

  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,      &B0,
                        /* ** */   /* ** */
                                   &B1,
                           BB,     &B2,    1, FLA_BOTTOM );
    /*-----------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                           /* ** */   /* ** */
                             &BB,       B2,    FLA_TOP );
  }
}
```

```
void FLASH_Gemm_nt( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                 ...                             )
   Assumption: A is a matrix, B is a row of blocks (row panel),
               C is a column of blocks (column panel) */
{
  FLA_Obj AT,     A0,     CT,     C0,
          AB,     A1,     CB,     C1,
                  A2,     C2;

  FLA_Part_2x1( A,    &AT,
                      &AB,       0, FLA_TOP );
  FLA_Part_2x1( C,    &CT,
                      &CB,       0, FLA_TOP );

  while ( FLA_Obj_length( AT ) < FLA_Obj_length( A ) ){

    FLA_Repart_2x1_to_3x1( AT,           &A0,
                        /* ** */      /* ** */
                                      &A1,
                           AB,        &A2,
                           1, FLA_BOTTOM );
    FLA_Repart_2x1_to_3x1( CT,           &C0,
                        /* ** */      /* ** */
                                      &C1,
                           CB,        &C2,
                           1, FLA_BOTTOM );
    /*-----------------------------------------------*/
    FLASH_Gepp( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                alpha, A1,     B,
                beta,  C1 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &AT,           A0,
                                             A1,
                           /* ** */      /* ** */
                             &AB,           A2,     FLA_TOP );
    FLA_Cont_with_3x1_to_2x1( &CT,           C0,
                                             C1,
                           /* ** */      /* ** */
                             &CB,           C2,     FLA_TOP );
  }
}
```

```
void FLASH_Gepp_nt( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
/* Special case with mode parameters
     FLASH_Gepp( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                 ...                             )
   Assumption: C is a block and
               A, C are rows of blocks (row panels) */
{
  FLA_Obj AL,    AR,       A0,  A1,  A2,
          BL,    BR,       B0,  B1,  B2;

  FLA_Part_1x2( A,    &AL,  &AR,      0, FLA_LEFT );

  FLA_Part_1x2( B,    &BL,  &BR,      0, FLA_LEFT );

  FLA_Scal( beta, FLASH_MATRIX_AT( C ) );

  while ( FLA_Obj_width( AL ) < FLA_Obj_width( A ) ){

    FLA_Repart_1x2_to_1x3(
       AL,  /**/ AR,        &A0, /**/ &A1, &A2,
       1, FLA_RIGHT );

    FLA_Repart_1x2_to_1x3(
       BL,  /**/ BR,        &B0, /**/ &B1, &B2,
       1, FLA_RIGHT );
    /*-----------------------------------------------*/
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                alpha,   FLASH_MATRIX_AT( A1 ),
                         FLASH_MATRIX_AT( B1 ),
                FLA_ONE, FLASH_MATRIX_AT( C ) );
    /*-----------------------------------------------*/
    FLA_Cont_with_1x3_to_1x2(
       &AL,  /**/ &AR,        A0, A1, /**/ A2,
       FLA_LEFT );

    FLA_Cont_with_1x3_to_1x2(
       &BL,  /**/ &BR,        B0, B1, /**/ B2,
       FLA_LEFT );
  }
}
```

Figure 3: FLASH Implementation of the routines appearing in `FLASH_Chol_by_tiles_var3`.

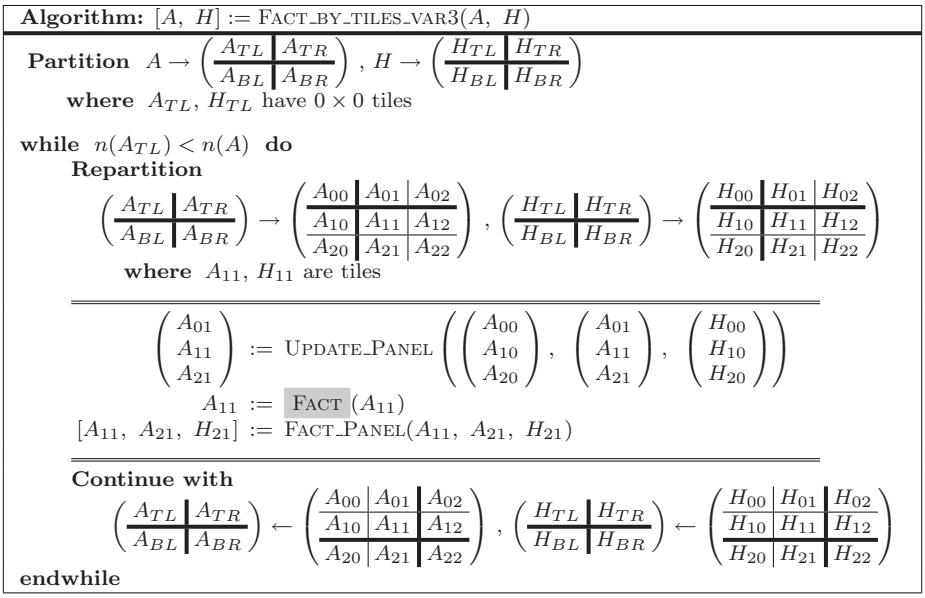


Figure 4: Left-looking algorithm-by-tiles for the LU and QR factorizations: General algorithmic framework. In the notation, $n(\cdot)$ stands for the number of columns of a matrix.

only recently. In [14], the authors introduce an algorithm-by-tiles for the QR factorization with the same numerical properties and practical computational cost as the slab-oriented version of the QR factorization. This framework was showed to render an algorithm-by-tiles for the computation of the LU factorization as well, with the same practical computational cost as the slab-oriented version of this factorization and similar numerical properties [19, 22]. The key to this algorithm is a careful organization of the sequence of row permutations during the LU factorization, combining the LINPACK and LAPACK styles of pivoting. These two algorithms-by-blocks are at the basis of the current implementations of these operations for multithreaded platforms in the FLAME run-time SuperMatrix [9, 23], and have been also adopted in the PLASMA project [8, 7].

Both algorithms-by-tiles reflect the same algorithmic framework, shown in Figures 4–6. Upon completion, the algorithm overwrites the upper triangular part of A with the upper triangular factor resulting from the LU/QR factorization. The strictly lower triangular part of A plus the auxiliary matrix H contain information on the Gaussian/Householder transforms employed during the factorization. As corresponds to a left-looking variant, at each iteration, algorithm FACT_BY_TILES_VAR3 first updates the tiles in the current panel (column of tiles) with respect to the factorization of the tiles to its left (routines UPDATE_PANEL and UPDATE_BLOCK; see Figure 5). Next, the LU/QR factorization of the current diagonal tile A_{11} is computed (kernel FACT); and, finally, the current iteration is completed with the factorization of the subdiagonal tiles in the current panel (routine FACT_PANEL; see Figure 6).

In Figures 4–6, we have highlighted four basic building kernels: FACT, APPLYT, FACT_{TD}, and $\text{APPLYT}_{\text{TD}}$. The first one corresponds to the basic factorization algorithm for the LU/QR factorization of a block, A_{11}, and the second one to the application of the Gaussian/Householder transforms obtained during the factorization of A_{11} to a second matrix. Thus, in the LU/QR factorization, kernel FACT is used to decompose tile A_{11} as $\bar{P}A_{11} = \bar{L}\bar{U}/A_{11} = \bar{Q}\bar{R}$, while kernel APPLYT computes $\bar{B} := \bar{P}^{\text{T}}\bar{L}^{-1}\bar{B}/\bar{B} := \bar{Q}^{\text{T}}\bar{B}$. The other two kernels yield analogous factorizations and updates associated with a matrix composed of 2×1 tiles, $\begin{pmatrix} \bar{B} \\ \bar{D} \end{pmatrix}$, with the top tile $\bar{B}$ being upper triangular. The careful implementation of these two kernels is crucial to yield a practical, high-performance implementation of the overall procedure. We will come back to this issue in

Algorithm: $[B] := \text{UPDATE\_PANEL}(A, B, H)$

Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$, $B \rightarrow \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right)$, $H \rightarrow \left(\begin{array}{c|c} H_{TL} & H_{TR} \\ \hline H_{BL} & H_{BR} \end{array}\right)$

where A_{TL}, H_{TL} have 0×0 tiles, B_T is 0 tiles high

while $n(A_{TL}) < n(A)$ **do**

Repartition

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \rightarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} H_{TL} & H_{TR} \\ \hline H_{BL} & H_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} H_{00} & H_{01} & H_{02} \\ \hline H_{10} & H_{11} & H_{12} \\ \hline H_{20} & H_{21} & H_{22} \end{array}\right)$$

where A_{11}, B_1, H_{11} are tiles

$B_1 := \text{APPLYT}(A_{11}, B_1)$

$[B_1, B_2] := \text{UPDATE\_BLOCK}(B_1, A_{21}, B_2, H_{21})$

Continue with

$$\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right), \quad \left(\begin{array}{c} B_T \\ \hline B_B \end{array}\right) \leftarrow \left(\begin{array}{c} B_0 \\ \hline B_1 \\ \hline B_2 \end{array}\right),$$

$$\left(\begin{array}{c|c} H_{TL} & H_{TR} \\ \hline H_{BL} & H_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} H_{00} & H_{01} & H_{02} \\ \hline H_{10} & H_{11} & H_{12} \\ \hline H_{20} & H_{21} & H_{22} \end{array}\right)$$

endwhile

Algorithm: $[C, E] := \text{UPDATE\_BLOCK}(C, D, E, H)$

Partition $D \rightarrow \left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right)$, $E \rightarrow \left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right)$, $H \rightarrow \left(\begin{array}{c} H_T \\ \hline H_B \end{array}\right)$

where D_T, E_T, H_T are 0 tiles high

while $m(D_T) < m(D)$ **do**

Repartition

$$\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \rightarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right), \quad \left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right) \rightarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ \hline E_2 \end{array}\right), \quad \left(\begin{array}{c} H_T \\ \hline H_B \end{array}\right) \rightarrow \left(\begin{array}{c} H_0 \\ \hline H_1 \\ \hline H_2 \end{array}\right)$$

where D_1, E_1, H_1 are tiles

$$\left(\begin{array}{c} C \\ \hline E_1 \end{array}\right) := \text{APPLYT}_{\text{TD}} \left(\left(\begin{array}{c} H_1 \\ \hline D_1 \end{array}\right), \left(\begin{array}{c} C \\ \hline E_1 \end{array}\right) \right)$$

Continue with

$$\left(\begin{array}{c} D_T \\ \hline D_B \end{array}\right) \leftarrow \left(\begin{array}{c} D_0 \\ \hline D_1 \\ \hline D_2 \end{array}\right), \quad \left(\begin{array}{c} E_T \\ \hline E_B \end{array}\right) \leftarrow \left(\begin{array}{c} E_0 \\ \hline E_1 \\ \hline E_2 \end{array}\right), \quad \left(\begin{array}{c} H_T \\ \hline H_B \end{array}\right) \leftarrow \left(\begin{array}{c} H_0 \\ \hline H_1 \\ \hline H_2 \end{array}\right)$$

endwhile

Figure 5: Left-looking algorithm-by-tiles for the LU and QR factorizations: Update the tiles in current panel w.r.t. factorizations of tiles to its left.

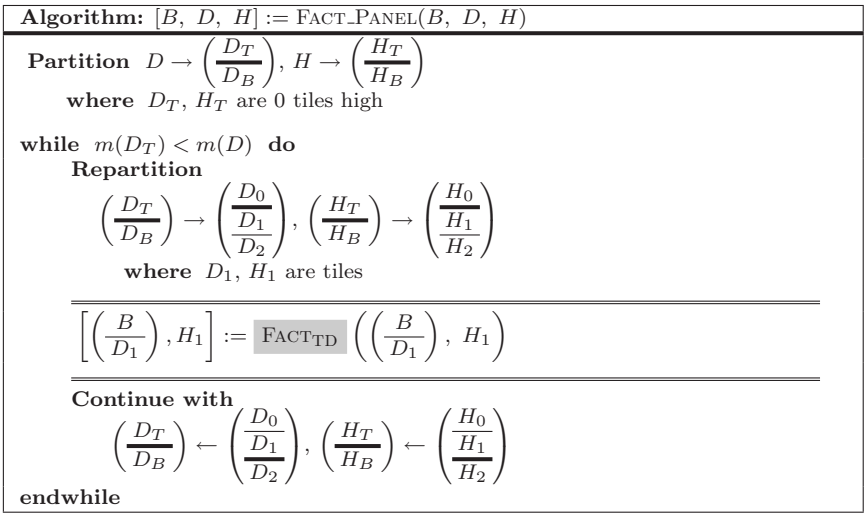


Figure 6: Left-looking algorithm-by-tiles for the LU and QR factorizations: Factorization of tiles in current panel.

Subsection 4.3, when we describe the parallelization of the OOC algorithms in multithreaded architectures.

To further illustrate the operation of the algorithms-by-tiles for the LU/QR factorizations and the effect of the basic kernels, consider the following partitionings of matrices A and H into 9 tiles each, forming a two 3×3 grids of tiles:

$$A \rightarrow \begin{pmatrix} T_{00} & T_{01} & T_{02} \\ T_{10} & T_{11} & T_{12} \\ T_{20} & T_{21} & T_{22} \end{pmatrix}, \qquad H \rightarrow \begin{pmatrix} L_{00} & L_{01} & L_{02} \\ L_{10} & L_{11} & L_{12} \\ L_{20} & L_{21} & L_{22} \end{pmatrix}.$$

Then, the sequence of invocations to the basic kernels FACT, APPLYT, FACT$_{\text{TD}}$, and APPLYT$_{\text{TD}}$ is given in Figure 7.

For simplicity, in the previous presentations we did not make explicit the use of incremental pivoting during the computation of the LU factorization; see [22] for details. Information on the size and structure of the auxiliary matrix H can also be found there.

3 OOC Algorithms

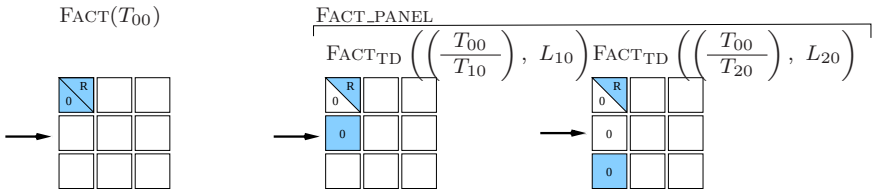
We next show how the algorithm-by-tiles form the basis of the high-performance OOC implementation of a dense matrix operation on multithreaded architectures.

3.1 Traditional OOC algorithms and algorithms-by-tiles

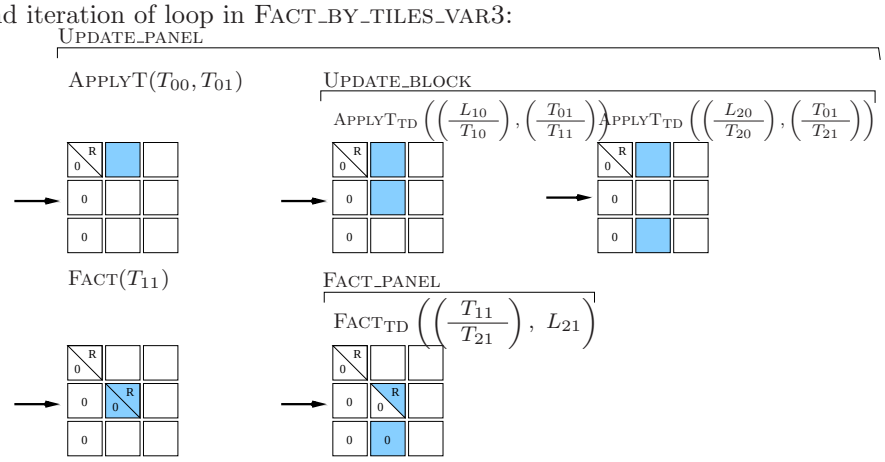
In conventional OOC implementations of algorithms-by-tiles, a small number of tiles (usually, between 1 and 3) are kept in-core at any moment, and the tile size is set so as to occupy as much as possible of the memory. Using large tiles aims at amortizing the cost of I/O with the number of operations these tiles are involved in ($O(t^2)$ memory operations –memops– vs. $O(t^3)$ floating-point arithmetic operations –flops, respectively).

In order to transform the algorithms-by-tiles in Figures 2–3 into OOC implementations, first we need to decide the maximum number of tiles that will be kept in-core during the execution of the algorithm, and then insert the necessary I/O calls in their codes. For example, a three-tile OOC algorithm for the Cholesky factorization is obtained by introducing the modifications in the codes corresponding to the algorithms-by-tiles shown in Figure 8. There, routine `FLASH_OOC_to_INC` is used to retrieve a tile from disk and `FLASH_INC_to_OOC`

First iteration of loop in FACT_BY_TILES_VAR3:



Second iteration of loop in FACT_BY_TILES_VAR3:



Third iteration of loop in FACT_BY_TILES_VAR3:

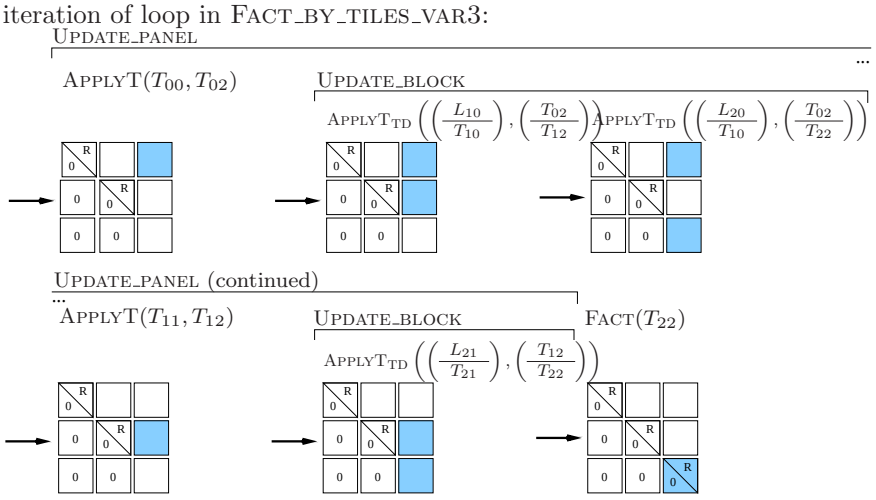


Figure 7: Operation of the algorithm-by-tiles for the LU and QR factorization and effect of the four basic building blocks, when processing a matrix composed of 3×3 tiles of dimension $t \times t$ each. Only those blocks highlighted are modified during the corresponding operation.

```
FLA_Error FLASH_Chol_by_tiles_var3( FLA_Obj A, int nb_alg )
{
  ...
    /*-----------------------------------------------*/
->  FLASH_OOC_to_INC( A11, Ainc );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A10,
->              FLA_ONE,       Ainc );
->  FLA_Chol_blk_var1( FLASH_MATRIX_AT( Ainc ), nb_alg );
->  FLASH_INC_to_OOC( Ainc, A11 );
    FLASH_Gemp( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                FLA_MINUS_ONE, A20, A10,
                FLA_ONE,       A21 );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    /*-----------------------------------------------*/
  ...
}
```

```
void FLASH_Syrk_ln( FLA_Obj alpha, FLA_Obj A,
                    FLA_Obj beta,  FLA_Obj C )
{
  ...
    /*-----------------------------------------------*/
->  FLASH_OOC_to_INC( A1, Ainc );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
->            alpha, FLASH_MATRIX_AT( Ainc ),
              beta,  FLASH_MATRIX_AT( C ) );
    /*-----------------------------------------------*/
  ...
}
```

```
void FLASH_Gepp_nt( FLA_Obj alpha, FLA_Obj A,
                                   FLA_Obj B,
                    FLA_Obj beta,  FLA_Obj C )
{
  ...
->FLASH_OOC_to_INC( C1, Cinc );
->FLA_Scal( beta, FLASH_MATRIX_AT( Cinc ) );

  while ( FLA_Obj_width( AL ) < FLA_Obj_width( A ) ){
    ...
    /*-----------------------------------------------*/
->  FLASH_OOC_to_INC( A1, Ainc );
->  FLASH_OOC_to_INC( B1, Binc );
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
->              alpha,   FLASH_MATRIX_AT( Ainc ),
->                       FLASH_MATRIX_AT( Binc ),
->              FLA_ONE, FLASH_MATRIX_AT( Cinc ) );
    /*-----------------------------------------------*/
    ...
  }
->FLASH_INC_to_OOC( Cinc, C1 );
  ...
}
```

```
void FLASH_Trsm_rltn( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
{
  ...
->FLASH_OOC_to_INC( L, Linc );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ){
    ...
    /*-----------------------------------------------*/
->  FLASH_OOC_to_INC( B1, Binc );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
->            alpha, FLASH_MATRIX_AT( Linc ),
->                   FLASH_MATRIX_AT( Binc ) );
->  FLASH_INC_to_OOC( Binc, B1 );
    /*-----------------------------------------------*/
    ...
  }
  ...
}
```

Figure 8: Modifications in the loop bodies to transform the algorithm-by-tiles for Variant 3 of the Cholesky factorization and kernels used therein to obtain OOC implementations with explicit I/O.

to store it on disk; for simplicity, we hide the memory allocation and release that would be needed to obtain a true three-tile OOC implementation.

4 The Run-Time based Approach

4.1 View memory as a cache operated by a (software) run-time

As illustrated by the example in the previous section, traditional OOC algorithms employ the memory as a cache for data on disk that is explicitly managed by the programmer, who is responsible for orchestrating the transfers between memory and disk. The cache is configured with a very small number of lines (between 1 and 4), with each line capable of holding one large tile. The main reason for this is that the disk is much slower than the memory and, therefore, the use of large tiles is assumed to be necessary to amortize the cost of moving tiles between memory and disk with the amount of operations performed on these data. (Alternatively, I/O latency can be hidden by employing asynchronous I/O (*double-buffering*); however, this approach complicates the task of the developer of OOC algorithms.)

We propose a solution with two major differences from the traditional implementation:

1. In our approach, the cache is operated by a *run-time* that is responsible for moving data between the cache (main memory) and the disk, unburdening the programmer from this task. This implies that there is no need to modify the algorithms-by-tiles (the codes in Figures 2–3) to include explicit calls to the transfer routines (`FLASH_OOC_to_INC` and `FLASH_INC_to_OOC`), as the run-time makes it so that the programmer can be oblivious to the movement of tiles between disk and memory.

2. We view the cache as consisting of a few dozens of lines so that each tile is now smaller. Although unconventional when compared with traditional OOC algorithms, note that our view is akin to hardware caches, which also consist of a moderate number of small-size lines. On the other hand, our software

implementation of the cache accommodates a much higher degree of flexibility, as different features of the software cache (like, e.g., number of lines, replacement policy, mapping policy, etc.) can be easily tuned.

Although operating with tiles of smaller dimension may expose some I/O latency, in the next subsection we will show how to avoid this by using asynchronous I/O operated by the run-time system, so that the programmer is also unburden from this task.

Following a classical text in Computer Architecture [15], we answer the following three questions to define the configuration of our software cache:

1. Where is a tile placed in the cache? We use a set associative cache configuration. A set is a group of lines in the cache. A tile is first mapped onto a set and it can be placed anywhere within that set. Compared with a cache with direct mapping, associative caches reduce the number of misses at the cost of a more complex management. However, given that we are operating with large tiles, the overhead implied by an associative organization is negligible when compared with the transfer times.

2. Which tile should be replaced on a cache miss? We employ the *least-recently used* (LRU) replacement policy.

3. What happens on a write? We implement a *write-back* policy: the information is written only to the tile in the cache. Modified cache tiles are written to disk only when they are replaced.

The run-time can be encoded in the form of a *software preamble* that is executed at the beginning of each kernel of the form "`FLA_`" (e.g., when executing routines `FLASH_Chol_by_tiles_var3` and the corresponding building blocks `FLASH_Syrk`, `FLASH_Trsm`, `FLASH_Gemm` and `FLASH_Gepp`, the preamble is run each time one of the kernels `FLA_Chol_blk_var1`, `FLA_Syrk`, `FLA_Gemm` and `FLA_Trsm` is encountered). For each tile parameter of these kernels, this piece of code first checks whether the data already resides in the (corresponding set of the) cache. If that is not the case, the preamble determines which tile from the (appropriate set of the) cache is evicted to make place for the new tile (following the LRU policy). If the tile that needs to leave the cache is dirty (i.e., it was modified by a previous operation), then it is written to disk; otherwise, its contents are simply replaced by the new tile. Thus, the mechanism that operates the software cache is completely hidden in the calls to the `libflame` implementation of BLAS.

We next illustrate the operation of the run-time using the factorization of a matrix A consisting of 4×4 tiles of dimension $t \times t$ each:

$$A \rightarrow \begin{pmatrix} T_{00} & T_{01} & T_{02} & T_{03} \\ T_{10} & T_{11} & T_{12} & T_{13} \\ T_{20} & T_{21} & T_{22} & T_{23} \\ T_{30} & T_{31} & T_{32} & T_{33} \end{pmatrix}.$$

Consider, e.g., the sequence of routines (left) and corresponding kernels (right) invoked during second iteration of routine `FLASH_Chol_by_tiles_var3`:

```
(a)
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A10,
                FLA_ONE,       A11 );
(b) /*-----------------------------------------------*/
    FLA_Chol_blk_var1( FLASH_MATRIX_AT( A11 ), nb_alg );
(c) /*-----------------------------------------------*/
    FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
                FLA_MINUS_ONE, A20, A10,
                FLA_ONE,       A21 );




(e) /*-----------------------------------------------*/
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
```

```
(a)
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, T10,
              FLA_ONE,       T11 );
(b) /*-----------------------------------------------*/
    FLA_Chol_blk_var1( T11, nb_alg );
(c) /*-----------------------------------------------*/
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
              FLA_MINUS_ONE, T20, T10,
              FLA_ONE,       T21 );
(d)
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
              FLA_MINUS_ONE, T30, T10,
              FLA_ONE,       T31 );
(e) /*-----------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, T11,
                       T21 );
(f)
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, T11,
                       T31 );
(g)
```

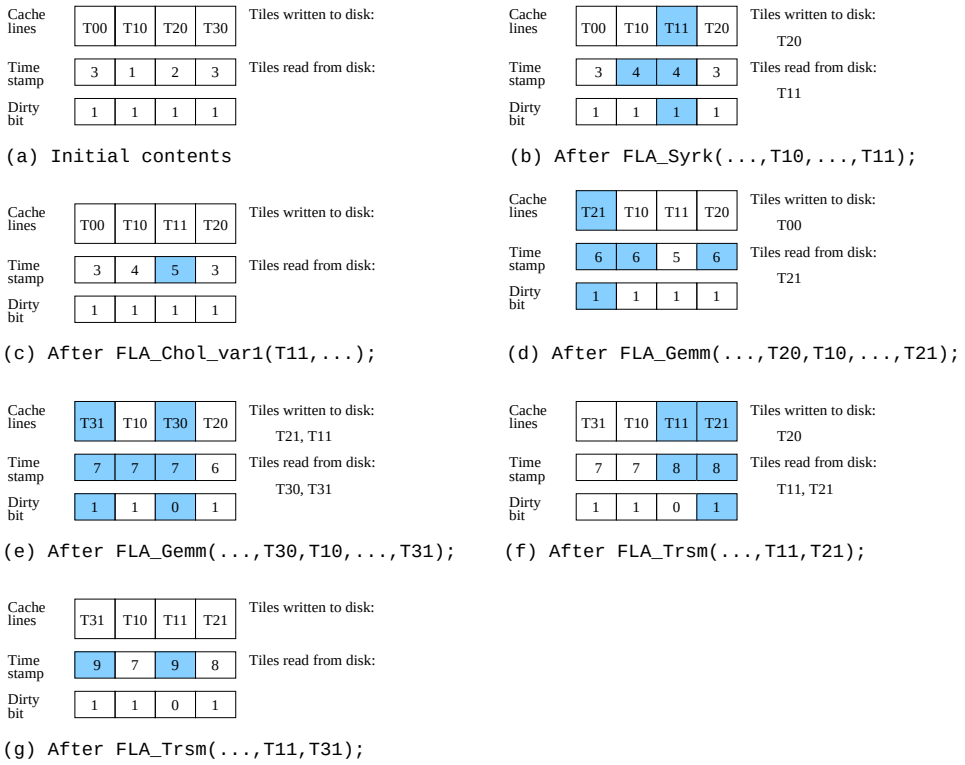


Figure 9: Evolution of the cache system during the second iteration of routine `FLASH_Chol_by_tiles_var3`.

and assume that the run-time operates on a fully associative cache (there is only one set) consisting of four lines (storage for four tiles) and with the initial contents shown in Figure 9 (a) (top left corner). There, the time stamp indicates the order in which tiles where accessed (a smaller value means older) and the dirty bit specifies whether the contents of the line have been modified ("0/1"=unmodified/dirty), and, therefore, need to be written to disk if evicted from the cache.

The evolution of the system, at each one of the points identified as `(b)`, `(c)`, `(d)`,... in the previous sequence of routines/kernels is then shown in Figure 9. Changes in the system with respect to the previous situation are highlighted in the figure.

4.2 Asynchronous I/O operated by a run-time

The problem with the cache-based system that we have described is that, by reducing the tile size, the time to retrieve a $t \times t$ tile from disk ($O(t^2)$ memops) may become non-negligible compared with the cost of operating with it ($O(t^3)$ flops). We next describe how to hide I/O latency also for tiles of moderate dimension by relying on asynchronous I/O. The novelty of our approach is that we advocate for a separation of concerns so that the programmer/developer of dense linear algebra codes remains completely isolated from the asynchronous mechanism, which is handled by the same run-time that operates the software cache.

Many dense linear algebra blocked algorithms and algorithms-by-blocks (among them, those for the Cholesky, QR and LU factorizations) share the property that once the block size is fixed, the sequence of operations that have to be performed on the (blocks of) data is completely determined. This is an important observation that we already exploited in the formulation of the SuperMatrix run-time scheduler for dense linear algebra codes on multithreaded architectures [23]. Here we will exploit this property differently, with the goal of implementing an accurate asynchronous I/O prefetch engine. Hardware prefetch retrieves

elements in advance based on data locality (when an element is accessed, the hardware also prefetches those elements around it into the cache). Compiler-controlled prefetch strategies insert instructions to schedule prefetches with execution, usually by unrolling loops. Our prefetch mechanism combines features of these two approaches: it is active at execution time, like hardware prefetch, but it is based on loop unrolling to prefetch tiles involved in future operations, like compiler-based prefetch.

Let us describe how we incorporate asynchronous I/O into the run-time system. There are two stages during the execution of an OOC implementation of an algorithm-by-tiles in our system. In the first stage, the run-time does a symbolic execution of the code creating a *list of pending tasks* (i.e., tasks that need to be executed to compute the desired operation). This is achieved by linking the user's application with a library containing special implementations of kernels "`FLA_`" which, instead of executing the corresponding operation, just create a new entry in the list of pending tasks. The entries of the list specify the task to be computed and the operands (tiles) which are involved. This process is equivalent to fully unrolling the loops in the codes: Upon completion of this first stage, the pending list contains the complete list of tasks, in the same order as they are encountered in the codes. Note that, when operating with large dense linear algebra problems, the cost of creating this list is negligible compared with the cost of executing the tasks in the list.

During the second stage, the real I/O and computation commence. The run-time now operates with the information contained in the list of pending tasks and no reference to the application code is necessary. Here a *scout* thread and a *worker* thread collaborate to perform, respectively, I/O and computations, thus implementing the asynchronous I/O mechanism. In particular, the scout thread extracts the first task from the pending list, and checks whether the tiles for this task are already in the software cache (memory), retrieving them from disk otherwise. Once all data (tiles) are in-core for a given task, the scout thread moves the corresponding entry from the pending list to the *list of ready tasks*, which only contains operations with all data in memory. The scout thread repeats this cycle until the pending list is empty, stopping in case it has to synchronize with the worker thread (discussed later).

The worker thread continuously inspects the ready list for tasks to be executed till there appears one new entry in the list (busy-wait). It then extracts this task from the list and executes the corresponding operation. As the tiles involved by the operation are ensured to be already in-core, the worker thread needs not concern itself with I/O.

Now, consider how these two threads synchronize. The collaboration between them can be viewed as a classical buffered producer-consumer problem. The scout thread produces items (tasks with data in-core) that the worker thread consumes (by executing the corresponding operation), and the list of ready tasks plays the role of the buffer. The worker thread blocks when the buffer is empty (there are no ready tasks). How to prevent the scout thread from overflowing the buffer is more delicate: although the ready list could potentially grow to contain all tasks that need to be executed, a condition on tasks inserted in the ready list is that the corresponding data are in-core. Thus, as in practice the capacity of the software cache is limited, a regulation mechanism is also needed to pace the scout thread. The strategy that we have designed and incorporated into the run-time system works as follows: For each tile in the software cache, we include a counter of the number of operations in the ready list that refer to it. When the scout thread moves a task from the pending list to the ready list, it increments the counter by 1 for each one of the tiles involved in the corresponding operation which are already in the cache, or brings the tile in-core and sets its counter to 1. When the worker thread completes the execution of a task in the ready list, it decrements the counter by 1 for each one of the tiles involved in the corresponding operation. Tiles in the software cache with a value of this counter greater than 0 are not eligible for being evicted from the cache, as there is surely one or more tasks (in the ready list) that will use them in the near future. The LRU policy selects among those tiles for which the counter equals 0 (taking into account also criteria like associativity), moving dirty data to disk or overwriting unmodified data with the new tile(s), as dictated by the write-back policy. If no tiles can be evicted from the cache, the scout thread blocks in a busy-wait until the worker thread completes the execution of some ready operations.

Proceeding in this manner yields a synchronization mechanism with a perfect prefetch: the scout thread knows exactly which tiles need to be moved from disk to memory by looking into the list of pending tasks. The result is a perfect prefetch engine, with a 100% hit rate. The mechanism can be further refined: e.g., if there are multiple candidate tiles to be evicted from the cache, the scout thread can look into the pending

list to select among those, implementing a policy that we could name as *not-needed-soon*.

4.3 Extracting parallelism for multithreaded architectures

Due to the presence of the run-time, in the overview of the parallel codes presented in this subsection we will consider that data ($t \times t$ tiles) are already in core.

General approach We propose two approaches for the parallelization of the OOC algorithms-by-tiles in a platform equipped with multiple cores, with a thread per core in both cases. The first approach exploits the coarse-grain parallelism existing among operations (tasks) in the ready list. Although operations are inserted in this list in the order they are encountered in a sequential symbolic execution of the code, any two operations in the list can be executed concurrently (or even reordered) provided there are no (direct or indirect) data dependencies between them. Therefore, we could employ a collection of worker threads that inspect the ready list and concurrently execute operations appearing in it, provided dependencies are preserved. Although this was exploited in [23] in the context of a parallel execution of in-core dense linear algebra operations in general-purposed multi-core processors, there is an important difference that made us abandon this option in the OOC case. In particular, the existence of multiple threads executing different tasks/operations will stress the I/O system, which may not be fast enough to feed them. The use of a unique software cache shared by several worker threads in combination with the run-time scout thread will require the use of smaller tiles, to maintain a long list of ready tasks (without dependencies) that expose enough parallelism to feed all cores. However, by reducing the tile size, the cost of I/O may become relevant, even when asynchronous I/O is in place. Using a distributed software cache, one per thread, also requires reducing the tile size, with similar negative consequences.

As an alternative, we propose a "fine"-grain parallelization approach, with the operations in the ready list executed in strict sequential order, but multiple worker threads collaborating in the execution of each operation. Therefore, we will next consider the isolated parallelization of the individual tasks that appear in the implementations of the algorithms-by-tiles for the Cholesky, LU and QR factorizations, as these are the operations (tasks) that appear in the ready list.

Cholesky factorization Let us start with the Cholesky factorization. There are four kernels involved the left-looking OOC algorithm-by-tiles for this operation: `FLA_Chol_blk_var1`, `FLA_Syrk`, `FLA_Trsm` and `FLA_Gemm`. Given $r = n/t$, the kernels are invoked, respectively, r, $r(r-1)/2$, $r(r-1)/2$ and $r^3/6 + O(r^2)$ times during the factorization. Thus, the performance of the matrix-matrix product will determine the global performance of the algorithm-by-tiles.

When the target architecture is a general-purpose multi-core architecture or a shared-memory multiprocessor, (like current platforms based on general-purpose processors from Intel, AMD or IBM,) in general high performance can be attained for kernels `FLA_Syrk`, `FLA_Trsm` and `FLA_Gemm` by using multithreaded implementations of BLAS from the hardware manufacturer (Intel MKL, AMD ACML or IBM ESSL). The efficient parallelization of the former kernel, `FLA_Chol_blk_var1`, is more delicate. One possibility to extract parallelism from this kernel is to employ the (right-looking BLAS-3 based) LAPACK legacy code `_potrf` linked with a multithreaded implementation of BLAS; and better results can be obtained by employing a multithreaded implementation of the LAPACK routine in a library like MKL. However, due to the complex pattern of dependencies of this operation, in general these two solutions deliver suboptimal performance. An alternative we choose here is to employ an algorithm-by-blocks to compute this operation, combined with the SuperMatrix run-time scheduler.

When the platform is also equipped with a many-core GPU, we rely on a native implementation (from the hardware vendor) of the matrix-matrix product to execute kernels `FLA_Syrk`, `FLA_Trsm` and `FLA_Gemm` in the hardware accelerator. For example, NVIDIA CUBLAS provides highly efficient single-precision codes for the matrix-matrix product on its G80 and G200 graphics processors, while the symmetric rank-t update and the triangular system solve can be built on top of it [2, 18]. On the other hand, in order to compute the Cholesky factorization of a tile, we propose to use the hybrid CPU-GPU implementation of `FLA_Chol_blk_var1` described in [3]. We will demonstrate that, for the tile sizes that we consider in our experiments, the transfer

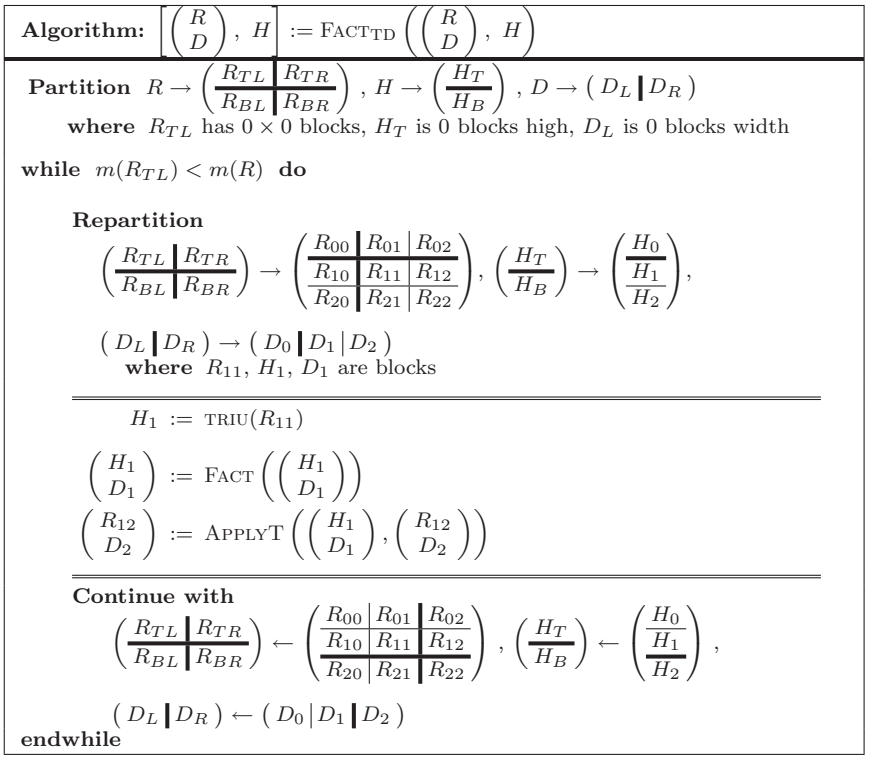


Figure 10: Structure-aware algorithm-by-blocks for the factorization of a matrix composed of 2×1 tiles, with the top tile R being upper triangular. $\text{TRIU}(R)$ denotes the upper triangular part of R.

cost of moving the data between the RAM and the memory of the hardware accelerator is negligible.

LU and QR factorizations Consider next the parallelization of the kernels appearing in the algorithm-by-blocks for the LU and QR factorizations presented in Subsection 2.3: FACT, APPLYT, FACT_{TD} and $\text{APPLYT}_{\text{TD}}$. These kernels occur, respectively, r, $r(r-1)/2$, $r(r-1)/2$ and $r^3/3 + O(r^2)$ times during the algorithm so that it is the parallel efficiency of the latter that dictates the overall performance of the algorithm.

Kernel FACT presents an algorithmic dependency pattern much like that of the Cholesky factorization so that, in principle, we can also employ algorithms-by-blocks, for both the LU and QR factorizations, in combination with the run-time scheduler introduced in [23]. However, given the minor impact of this kernel on the global parallel performance of the OOC algorithm, we have preferred to use a standard parallel solution based on the multithreaded implementations of the LU and QR factorization kernels in MKL.

The parallel application of the transforms generated during the factorization of a general dense matrix, required by kernel APPLYT, is simple. Consider, e.g., the QR factorization $\bar{A} = \bar{Q}\bar{R}$, where $\bar{A}$ is one of the tiles of the matrix being factored and let $\bar{B}$ denote a second tile from this matrix. Kernel APPLYT is responsible for the computation $\bar{B} := \bar{Q}^T\bar{B}$, where $\bar{Q}$ is not explicitly formed but applied as a sequence of Householder transforms using the compact WY representation. Given that $\bar{B}$ has a large number of columns (in our experiments, $t \approx 5,000$) and the number of threads h is small compared with t, an efficient BLAS-3 parallelization of this kernel can be easily obtained by splitting tile $\bar{B}$ into t/h panels (blocks of columns) so that each thread is responsible for the application of all the Householder transforms to one of these panels.

The parallelization of kernel FACT_{TD}, which computes the factorization of matrix $\begin{pmatrix} \bar{R} \\ \bar{D} \end{pmatrix}$ with the top tile $\bar{R}$ being upper triangular, is more challenging. In particular, special care is needed to exploit/preserve

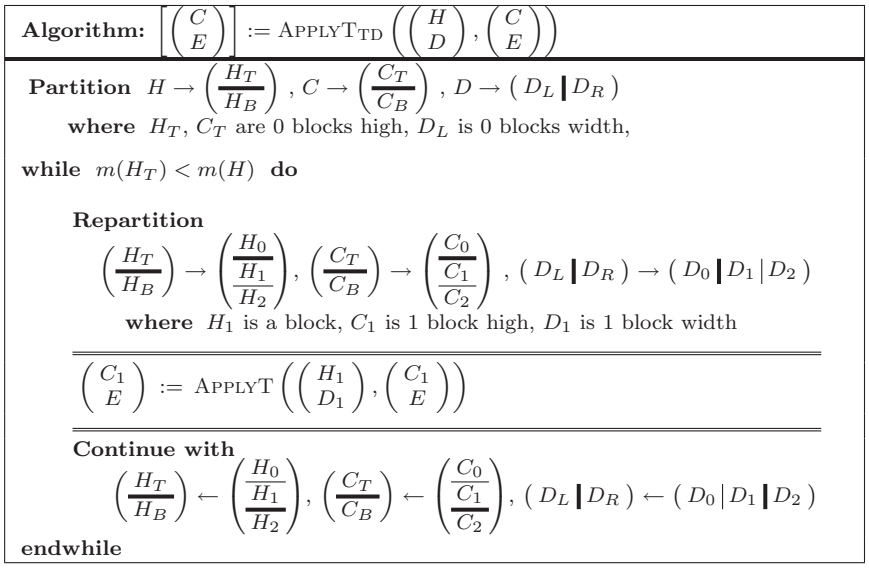


Figure 11: Structure-aware algorithm-by-blocks for the application of the transforms resulting from factorization of a matrix composed of 2×1 tiles, with the top tile R begin upper triangular.

the upper triangular structure of $\bar{R}$ as, otherwise, the cost of the algorithm-by-blocks becomes significantly higher than that of the standard LU and QR factorizations. Figure 10 illustrates the implementation of this kernel. Denoting $s = t/b$, FACT_{TD} invokes s times kernel FACT and $s(s-1)/2$ times kernel $\text{APPLYT}_{\text{TD}}$. Although we have discussed the parallel implementation of kernel FACT earlier, the dimensions of the data involved is much smaller now: blocks of size $b \times b$ compared with tiles of size $t \times t$. Therefore, we do not parallelize kernel FACT when invoked from FACT_{TD}. The strategy adopted to execute kernel APPLYT in parallel when invoked from FACT_{TD} is analogous to the one presented earlier: we split $\begin{pmatrix} R_{12} \\ D_2 \end{pmatrix}$ into h panels (provided the number of columns is large relative to h) and perform the application of the transforms in parallel.

The structure-preserving algorithm-by-blocks of kernel $\text{APPLYT}_{\text{TD}}$ is similar: see Figure 11. Here we split the $(b+t) \times t$ matrix $\begin{pmatrix} C_1 \\ E \end{pmatrix}$ into h panels and apply the transforms in $\begin{pmatrix} T_1 \\ D_1 \end{pmatrix}$ concurrently using h threads.

The above discussion of the parallelization of the four kernels appearing in the LU and QR factorizations is valid when the target architecture is a general-purpose multi-core platform. Unfortunately, kernel $\text{APPLYT}_{\text{TD}}$, which determines the parallel performance of the algorithm-by-tiles, is difficult to implement efficiently on a many-core GPU. In particular, the application of the transforms results in a fine-grained parallelism which is not well suited for GPUs. We will therefore not consider the implementation of GPU-accelerated versions of the OOC algorithms-by-tiles for the LU and QR factorizations.

5 Experiments

All the experiments in this section were performed on three different platforms:

- TESLA is a workstation with two Intel Xeon QuadCore E5405 processors (8 cores) at 2.0 GHz and 8 Gbytes of DDR2 RAM (single-precision peak performance is $128 \cdot 10^9$ flops per second; i.e., 128 GFLOPS). The Intel 5400 chipset provides an I/O interface with a peak bandwidth of 1.5 Gbits/second. The disk is a SATA-I (GB0160CAABV, 7200 r.p.m.) with a total capacity of 160 Gbytes.

- ZAPEG is a workstation with an AMD Phenom 9550 QuadCore processor at 2.2 GHz and 4 Gbytes of DDR2 RAM, connected to an NVIDIA GeForce 9800 GX2 GPU (single-precision peak performance is 70.4 GFLOPS for the AMD processor plus 546 GFLOPS for the GPU). The AMD chipset provides an I/O interface with a peak bandwidth of 3 Gbits/second. The system has two SATA-II (Seagate ST3160815AS, 7200 r.p.m.) disks, forming a RAID-0 of 2×160 Gbytes.

- TESLA2 is a workstation with two Intel Xeon QuadCore E5440 processors at 2.83 GHz and 16 Gbytes of DDR2 RAM (single-precision peak performance is 181.12 GFLOPS). The peak bandwidth of the I/O interface is 3 Gbits/second. The system has six SAS (Seagate ST3146356SS, 15.000 r.p.m.) disks in a RAID-0, for a total storage of 6×146 Gbytes. Connected to it is an NVIDIA Tesla S1070 (single-precision peak performance of 933 GFLOPS). In our experiments with this platform, to distinguish between the use of the Intel cores from the use of a single GPU in the Tesla S1070, we will refer to these cases as TESLA2 and TESLA2G, respectively.

Multithreaded implementations of BLAS (and LAPACK) kernels were provided in MKL 10.0.1 for both the Intel and AMD multi-core processors. The implementation of the matrix-matrix product in CUBLAS 2.0 and our own implementations of several other kernels built on top of it (see, e.g., [18]) were employed for the NVIDIA GPUs. Single-precision arithmetic was employed in all experiments. Although double-precision is the standard in numerical applications, single-precision can be particularly appealing as it reduces the I/O stress and an initial solution of a linear system with single-precision accuracy can be cheaply converted into a double-precision via iterative refinement [16]. Single-precision is also specially appealing for current hardware accelerators, which are highly biassed towards it.

5.1 Performance and tuning of basic kernels

We first analyze the performance of the in-core basic building kernels involved in the Cholesky, QR and LU factorizations. Tables 1 and 2 report the number of invocations (calls) to each basic kernel during the factorization of a square matrix of order n using the algorithms-by-tiles for the (left-looking) Cholesky, LU and QR factorizations with tile size t and $r = n/t$. In the tables, we also report the theoretical cost (with lower order terms neglected in the expressions) and the performance of the kernels, measured in terms of flops and GFLOPS, respectively. In the LU and QR factorizations we only count "useful computations" and do not consider additional operations that are artificially introduced in order to expose more parallelism and/or cast a larger fraction of the computations in terms of matrix multiplications. In this analysis we set t=5,120 which, in our experience, was found to be an fair value for the OOC algorithms-by-tiles. (Only in the experimentation with the Cholesky factorization on TESLA, larger tile sizes, t=7,680 and t=10,240, attained slightly higher performances for the largest problem sizes.) In the experiments we also found that, for the algorithms-by-blocks for the LU and QR factorizations in Figures 10 and 11, the block size b=64 was optimal in most cases.

Clearly, the performance of `FLA_Gemm` will determine the efficiency of the OOC algorithm for the Cholesky factorization while the efficiency of the OOC algorithms for the LU and QR factorization is conditioned by that of kernel APPLYT$_{\text{TD}}$. In other words, the efficiency of these kernels sets an upper bound to the GFLOPS rate that can be attained with the OOC codes, provided I/O latency is completely hidden. Whether these bounds are within reach will depend on the percentage of the flops that are cast in term of the corresponding kernel and the impact of I/O.

The two tables also illustrate the results of our efforts to improve the performance of the basic kernels. In Table 1, there are two columns that display GFLOPS rates on the GPU available at TESLA2G, labelled as "CUBLAS" and "UJI CUBLAS". The first one corresponds to the GFLOPS rate attained with the native implementation of the kernels in NVIDIA CUBLAS 2.0; the second column of results is attained using our own tuned implementation of these kernels, which cast most computations in terms of the matrix-matrix product [18]. Table 2 also shows two columns of performance data: "MTBLAS" corresponds to the parallel execution obtained by linking with the multithreaded implementation of BLAS in MKL; "COLPAR" is the performance yielded by the panelwise parallel application of the kernels (see Subsection 4.3).

Due to the higher performance of our parallel kernels (between 1.38 and 2.23 speed-up for the GPU kernels in UJI CUBLAS compared with the native CUBLAS; 1.25 and 1.50 gains from the COLPAR approach

Kernel	#calls	flops/call	GFLOPS				
			TESLA	TESLA2	ZAPEG	TESLA2G	
						CUBLAS	UJI CUBLAS
`FLA_Chol`	r	$t^3/3$	88	115	69	–	113
`FLA_Syrk`	$r(r-1)/2$	t^3	88	115	147	134	300
`FLA_Trsm`	$r(r-1)/2$	t^3	110	147	125	181	251
`FLA_Gemm`	$r^3/6 + O(r^2)$	$2t^3$	112	151	151	246	351

Table 1: Number of invocations and in-core performance of the basic kernels involved in the OOC algorithms-by-tiles for the Cholesky factorization operating on tiles of size t=5,120.

Kernel	#calls	flops/call	GFLOPS TESLA		GFLOPS TESLA2	
			MT BLAS	COLPAR	MTBLAS	COLPAR
LU factorization						
FACT	r	$2t^3/3$	27	–	37	–
APPLYT	$r(r-1)/2$	t^3	65	78	96	113
FACT_{TD}	$r(r-1)/2$	t^3	36	–	42	–
$\text{APPLYT}_{\text{TD}}$	$r^3/3 + O(r^2)$	$2t^3$	52	65	55	82
QR factorization						
FACT	r	$4t^3/3$	38	–	51	–
APPLYT	$r(r-1)/2$	$2t^3$	47	63	66	81
FACT_{TD}	$r(r-1)/2$	$2t^3$	41	–	57	–
$\text{APPLYT}_{\text{TD}}$	$r^3/3 + O(r^2)$	$4t^3$	52	66	73	95

Table 2: Number of invocations and in-core performance of the basic kernels involved in the OOC algorithms-by-tiles for the LU and QR factorizations operating on tiles of size t=5,120.

compared with the traditional multithreaded BLAS in MKL), we will employ these when evaluating the OOC routines in the next experiments.

5.2 Performance of OOC routines

Our second experiment compares the performance of several in-core and OOC algorithms for all three factorizations:

In-core MKL: The (in-core) multithreaded implementations of the factorization in MKL.

In-core algorithm-by-blocks: The (in-core) algorithms-by-blocks with parallelism extracted dynamically by the SuperMatrix run-time system [23].

OOC traditional: Three- and four-tile OOC implementations with explicit invocations to I/O routines. The tile size was set in this routine to t=5,120. Although, in theory, using a large tile size turns the cost of moving data between disk and main memory ($O(t^2)$ disk accesses) negligible compared with the computational cost ($O(t^3)$ flops), in our experimentation we found a large drop in the disk transfer rate for tiles of dimension larger than $5,120 \times 5,120$. (We experienced similar behavior for several other current desktop systems equipped with different disks.) In these routines, parallelism was extracted implicitly by linking with a multithreaded implementation of BLAS.

OOC with software cache: OOC implementation with a software cache in place operated by the run-time to reduce the number of I/O transfers. The cache occupies 50%–75% of RAM, and is organized following a set associative configuration with t=5,120. Parallelism is extracted as discussed in Subsection 4.3.

OOC with asynchronous I/O: OOC implementation with software cache and asynchronous I/O operated by the run-time. The configuration of the cache and the approach to extract parallelism match those of the previous case.

Figure 12 reports the performance of these routines measured in GFLOPS, with the usual count of $n^3/3$, $2n^3/3$ and $4n^3/3$ flops for the Cholesky, LU and QR factorizations of a square matrix of order n, respectively. (Experiments with the OOC algorithms-by-tiles for the QR factorization on non-square matrices offered similar results. We also note here that the algorithms-by-tiles for the LU and QR factorizations perform a larger number of flops than the corresponding in-core factorization routines in MKL and LAPACK; see, e.g., [14] for details.)

The results in the figure show that the best OOC results always correspond to the implementation that overlaps computation and transfers between RAM and disk (**OOC with asynchronous I/O**). The performance of this code for the Cholesky factorization reaches 96, 136, 143 and 262 GFLOPS on TESLA, TESLA2, ZAPEG and TESLA2G, respectively. This represents 85%, 90%, 94% and 74% of the performance of the basic kernel `FLA_Gemm` in each architecture. For the LU and QR factorizations, this variant achieves, respectively, 61 and 63 GFLOPS on TESLA and 81 and 91 GFLOPS on TESLA2, which varies between 93% and 98% of the kernel APPLYT$_{\text{TD}}$. The performance results for these two operations reveal that I/O is mostly hidden and the GFLOPS rates for the OOC algorithms are maintained as the problem size is increased, thus confirming the scalability of the solution.

To assess the benefits contributed by the use of the software cache, Table 3 shows the number of tiles read from or written to disk for the OOC algorithm-by-blocks with explicit I/O calls (**OOC traditional**) and the ones that employ a software cache (both **OOC with software cache** and **OOC with asynchronous I/O**). The results demonstrate that the software cache greatly reduces the number of tiles that are transferred between the disk and RAM.

Matrix size (square)	**OOC traditional** #reads	#writes	**OOC with sw. cache** #reads	#writes
Cholesky factorization				
51,200	394	100	130	55
92,160	661	144	326	78
LU factorization				
51,200	1,045	715	229	100
92,160	5,985	4,047	2,171	324
QR factorization				
51,200	1,045	715	229	100
92,160	5,985	4,047	2,171	324

Table 3: Disk accesses (in terms of number of tiles read/written) attained by the OOC algorithms-by-blocks.

5.3 Solution of large-scale problems

Table 4 reports the execution time required to compute the Cholesky, LU and QR factorization using the implementation **OOC with asynchronous I/O** on TESLA2 and TESLA2G. The results demonstrate that what would have been considered a very large problem only a few years ago was solved in a few hours.

6 Conclusions

This paper focuses attention on programmability and how it can be achieved for OOC computations without giving up performance. It shows how a separation of concerns can be leveraged to implement OOC algorithms. The FLASH extension of the FLAME API allows the library programmer to develop code without explicit management of I/O. At run time, the library generates a task list which is then handed to a run-time

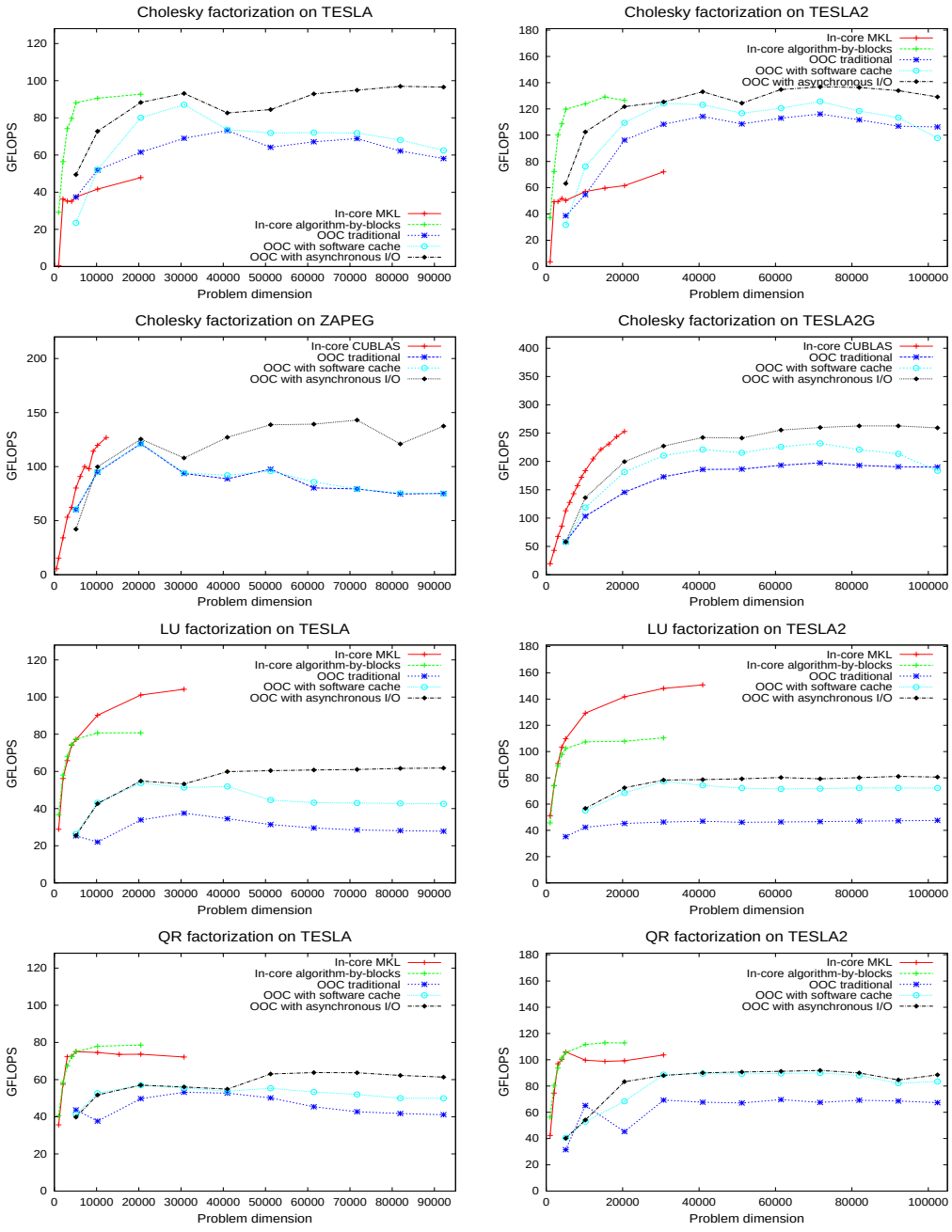


Figure 12: Performance of the codes for the Cholesky, LU and QR factorizations on a multi-core processor and a GPU.

Matrix size (square)	Time
Cholesky factorization on TESLA2 (8 Intel Xeon cores)	
102,400	56 min 16 sec
Cholesky factorization on TESLA2G (NVIDIA Tesla S1070)	
102,400	23 min 19 sec
LU factorization on TESLA2 (8 Intel Xeon cores)	
102,400	2 h 25 min 12 sec
QR factorization on TESLA2 (8 Intel Xeon cores)	
102,400	4 h 29 min 24 sec

Table 4: Execution time (in hours, minutes, and seconds) of the algorithms-by-tiles for the Cholesky, LU and QR factorizations using the OOC run-time with asynchronous I/O.

system that schedules tasks and data movement. Together, this provides a solution for the programmability problem in this domain. The present paper lays the foundation for extending these techniques to distributed memory architectures and new architectures like Intel SCC research processor [17] which has 48 cores on a single chip, a mixture of message-passing and shared memory model, and no cache coherency protocol.

Additional information

For further details on FLAME, visit

`http://www.cs.utexas.edu/users/flame/`.

Acknowledgments

Gregorio Quintana-Ortí, Francisco Igual, Mercedes Marqués and Enrique S. Quintana-Ortí were supported by the CICYT project TIN2008–06570-C04-01 and FEDER.

We thank the other members of the FLAME team for their support.

References

[1] M. Baboulin. *Solving large dense linear least squares problems on parallel distributed computers. Application to the Earth's gravity field computation.* Ph.D. dissertation, INPT, March 2006. TH/PA/06/22.

[2] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proceedings of the 10th IEEE Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2008*, pages CD–ROM, 2008.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurrency and Computation: Practice and Experience*, 21:2457–2477, 2009.

[4] Charles J. Bashe, Lyle R. Johnson, John H. Palmer, and Emerson W. Pugh. *IBM's early computers.* MIT Press, 1985.

[5] Natacha Bereux. Out-of-Core implementations of Cholesky factorization: Loop-based versus recursive algorithms. *SIAM Journal on Matrix Analysis and Applications*, 30(4):1302–1319, 2008.

[6] Paolo Bientinesi, John A. Gunnels, Margaret E. Myers, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. The science of deriving dense linear algebra algorithms. *ACM Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[7] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 190 UT-CS-07-600, University of Tennessee, September 2007.

[8] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.

[9] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007a. ACM.

[10] Po Geng, J. Tinsley Oden, and Robert van de Geijn. Massively parallel computation for acoustical scattering problems using boundary element methods. *Journal of Sound and Vibration*, 191(1):145–165, 1996.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 3rd edition, 1996.

[12] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[13] Brian C. Gunter. *Computational methods and processing strategies for estimating Earth's gravity field*. PhD thesis, The University of Texas at Austin, 2004.

[14] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Transactions on Mathematical Software*, 31(1):60–78, March 2005.

[15] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Pub., San Francisco, 2003.

[16] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

[17] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De1, R. Van Der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Proceedings of the International Solid-State Circuits Conference*, February 2010.

[18] Francisco D. Igual, Gregorio Quintana-Ortí, and Robert van de Geijn. Level-3 BLAS on a GPU: Picking the low hanging fruit. FLAME Working Note #37 DICC 2009-04-01, Depto. de Ingeniería y Ciencia de Computadores, Universidad Jaime I de Castellón, April 2009.

[19] Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. In *Proceedings of PARA 2004*, number 3732 in LNCS, pages 413–422. Springer-Verlag Berlin Heidelberg, 2005.

[20] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15, The University of Texas at Austin, Department of Computer Sciences, May 2004.

[21] J. L. Movilla, J. I. Climente, and J. Planelles. Dielectric polarization in axially-symmetric nanostructures: a computational approach. *Computer Physics Communications*, 181(1):92–98, 2010.

[22] Enrique Quintana-Ortí and Robert A. van de Geijn. Updating an LU factorization with pivoting. *ACM Transactions on Mathematical Software*, 35(2):11:1–11:16, July 2008.

[23] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*, 36(3):14:1–14:26, July 2009.

[24] Edwin D. Reilly. *Milestones in computer science and information technology.* Greenwood Press, Westport, CT, 2003. page 164.

[25] N. Schafer, R. Serban, and D. Negrut. Implicit integration in molecular dynamics simulation. In *ASME International Mechanical Engineering Congress & Exposition*, 2008. (IMECE2008-66438).

[26] Sivan Toledo and Fred G. Gustavson. The design and implementation of SOLAR, a portable library for scalable out-of-core linear algebra computation. In *Proceedings of IOPADS '96*, 1996.

[27] Yu Zhang and Tapan K. Sarkar. *Parallel solution of integral equation-based EM problems in the frequency domain.* John Wiley & Sons, Hoboken, NJ, 2009.