
Informe Técnico ICC 01-01-2009

# Solving “Large” Dense Matrix Problems on Multi-Core Processors and GPUs

Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. van de Geijn

Enero de 2009

Departamento de Ingeniería y Ciencia de Computadores

Correo electrónico: {mmarques, gquintan, quintana}@icc.uji.es;
rvdg@cs.utexas.edu

Universidad Jaime I
Campus de Riu Sec, s/n
12.071 - Castellón
España

# Solving “Large” Dense Matrix Problems on Multi-Core Processors and GPUs

Mercedes Marqués[1], Gregorio Quintana-Ortí[2], Enrique S. Quintana-Ortí[3], Robert A. van de Geijn[4]

**Abstract:** Few realize that, for large matrices, many dense matrix computations achieve nearly the same performance when the matrices are stored on disk as when they are stored in a very large main memory. Similarly, few realize that, given the right programming abstractions, coding Out-of-Core (OOC) implementations of dense linear algebra operations (where data resides on disk and has to be explicitly moved in and out of main memory) is no more difficult than programming high-performance implementations for the case where the matrix is in memory. Finally, few realize that on a contemporary eight core architecture or a platform equiped with a graphics processor (GPU) one can solve a $100,000 \times 100,000$ symmetric positive definite linear system in about one hour. Thus, for problems that used to be considered large, it is not necessary to utilize distributed-memory architectures with massive memories if one is willing to wait longer for the solution to be computed on a fast multithreaded architecture like a multi-core computer or a GPU. This paper provides evidence in support of these claims.

**Keywords:**
Dense linear algebra, Out-of-Core operations, multi-core processors, GPUs, high performance.

[1]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: mmarques@icc.uji.es.
[2]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: gquintan@icc.uji.es.
[3]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: quintana@icc.uji.es.
[4]Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
E-mail: rvdg@cs.utexas.edu.

# Solución de Problemas Matriciales de "gran Escala" sobre Procesadores Multinúcleo y GPUs

Mercedes Marqués[5], Gregorio Quintana-Ortí[6], Enrique S. Quintana-Ortí[7], Robert A. van de Geijn[8]

**Resumen:**

Pocos son conscientes de que, para matrices grandes, muchos cálculos matriciales obtienen casi el mismo rendimiento cuando las matrices se encuentran almacenadas en disco que cuando residen en una memoria principal muy grande. De manera parecida, pocos son conscientes de que, si se usan las abstracciones de programacin correctas, codificar algoritmos *Out-of-Core* (OOC) para operaciones de Álgebra matricial densa (donde los datos residen en disco y tienen que moverse explícitamente entre memoria principal y disco) no resulta más difícil que codificar algoritmos de altas prestaciones para matrices que residen en memoria principal. Finalmente, pocos son conscientes de que en una arquictura actual con 8 núcleos o un equipo con un procesador gráfico (GPU) es posible resolver un sistema lineal simétrico positivo definido de dimensión $100,000 \times 100,000$ aproximadamente en una hora. Así, para problemas que solían considerarse grandes, no es necesario usar arquitecturas de memoria distribuida con grandes memorias si uno está dispuesto a esperar un cierto tiempo para que la solución se obtenga en una arquitectura multihebra como un procesador multinúcleo or una GPU. Este trabajo presenta evidencias que soportan tales afirmaciones.

**Palabras clave:**

Álgebra lineal densa, operaciones *Out-of-Core*, procesadores multinúcleo, procesadores gráficos (GPUs), altas prestaciones.

[5]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: mmarques@icc.uji.es.

[6]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: gquintan@icc.uji.es.

[7]Departamento de Ingeniería y Ciencia de los Computadores
E-mail: quintana@icc.uji.es.

[8]Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
E-mail: rvdg@cs.utexas.edu.

# Solving "Large" Dense Matrix Problems on Multi-Core Processors and GPUs

Mercedes Marqués, Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
{mmarques,gquintan,quintana}@icc.uji.es

Robert A. van de Geijn
Department of Computer Sciences,
The University of Texas at Austin,
Austin, TX 78712. rvdg@cs.utexas.edu

## Abstract

*Few realize that, for large matrices, many dense matrix computations achieve nearly the same performance when the matrices are stored on disk as when they are stored in a very large main memory. Similarly, few realize that, given the right programming abstractions, coding Out-of-Core (OOC) implementations of dense linear algebra operations (where data resides on disk and has to be explicitly moved in and out of main memory) is no more difficult than programming high-performance implementations for the case where the matrix is in memory. Finally, few realize that on a contemporary eight core architecture or a platform equiped with a graphics processor (GPU) one can solve a* $100,000 \times 100,000$ *symmetric positive definite linear system in about one hour. Thus, for problems that used to be considered large, it is not necessary to utilize distributed-memory architectures with massive memories if one is willing to wait longer for the solution to be computed on a fast multithreaded architecture like a multi-core computer or a GPU. This paper provides evidence in support of these claims.*

## 1 Introduction

Examples of problems that require the solution of very large dense linear systems or linear least-squares problems include the estimation of Earth's gravitational field, Boundary Element formulations in electromagnetism and acoustics, and molecular dynamics simulations [2, 19, 17, 31, 38]. In these applications, *large* refers to matrices with a number of rows/columns in the $10^5$ to $10^7$ range. When these matrices become too large to fit in memory, one must either change the mathematical formulation of the problem or use secondary memory (e.g., disk). We will focus on the latter. While data stored on disk can be accessed via virtual memory, careful design of Out-of-Core (OOC) algorithms is generally required to attain high performance. Moreover, in the past cutting-edge architectures often did not incorporate virtual memory, which may happen again for future multi-core architectures particularly since virtual memory consumes considerable power. Thus, the topic of developing OOC algorithms for dense linear problems continues to be an active area of research.

There are only a few Open Source libraries available for OOC dense linear algebra computations. The traditional sequential LAPACK [1] library does not include OOC routines and the more recent `libflame` [37] library only includes prototype OOC capabilities [22]. For large-scale problems ScaLAPACK provides prototype OOC implementations of Cholesky, LU, and QR factorization based solvers [3, 9, 13] as does the SOLAR [33] library that builds upon ScaLAPACK routines for in-core computation. The Parallel Linear Algebra Package (PLAPACK) [34], which inspired the `libflame` sequential library, is an alternative to ScaLAPACK for message-passing architectures and provides an OOC extension, POOCLAPACK, that, like ScaLAPACK and SOLAR, targets message-passing architectures [30, 29, 20, 21, 38]. A survey on parallel OOC

implementations of individual operations and/or machine specific libraries for dense linear systems is given in [32]. Insights about how storing matrices by tiles (what we call blocks) facilitates scalability can be found in that paper.

In this paper,

- We briefly review the concept of algorithms-by-blocks and how an Application Programming Interface (API) developed as part of the FLAME project facilitates programming such algorithms. In brief, algorithms-by-blocks view matrices, possibly hierarchically, as a collection of submatrices (blocks) that become units of data. The algorithms then orchestrate the computation as operations with those blocks, which become units of computation.
- We review a run-time system, SuperMatrix, which given a linear algebra code constructs a Directed Acyclic Graph (DAG) of tasks (operations with blocks) and dependencies between tasks.
- We discuss how this approach can be extended by using the DAG to prefetch and/or cache data so that I/O is overlapped with computation transparently to the programmer.
- We report our experience with this approach on a platform that includes multiple cores and/or a graphics processor (GPU), and a RAM of moderate size, using the Cholesky factorization as a motivating example.
- We show that, once the problem size becomes large, the performance attained by the OOC implementation rivals that of a high-performance algorithm for matrices that fit in memory.
- We reason that this approach can also accommodate OOC implementations of algorithms-by-tiles for the level-3 Basic Linear Algebra Subprograms (BLAS) [15] and the LU and QR factorizations.
- We argue that the approach may become a highly cost-effective solution for solving these kinds of operations, making it possible for less well-funded projects to address medium to large size problems.

Together, these contributions advance the state-of-the-art in this area.

The rest of the paper is structured as follows. In Section 2 we review some of the fundamental parts of the FLAME project using the Cholesky factorization. An infrastructure in support of OOC computation is proposed in Section 3. Parallel execution of dense linear algebra operations with the data in-core is briefly addressed by using algorithms-by-blocks combined with dynamic scheduling and multi-threaded implementations of BLAS, as described in Section 4. Experiments reporting performance for the OOC Cholesky factorization on a multi-core platform and a workstation equiped with a GPU are reported in Section 5. We briefly indicate how a traditional application may interface with the proposed solvers in Section 6. We close the paper with a few concluding remarks and a brief discussion of future work in Section 7.

## 2 Cholesky Factorization using FLAME

Over the last decade we have developed a complete framework for fast and reliable generation of dense and banded libraries as part of the FLAME project (`http://www.cs.utexas.edu/users/flame`). The set of "tools" comprises a high-level notation for expressing algorithms for dense and banded linear algebra operations [18], a formal derivation methodology to obtain provably correct algorithms [7], high-level APIs to transform algorithms into codes [8], and a run-time system for the automatic parallelization of those codes on multi-core platforms; see [28] and the references therein. The result is a high-performance library for dense linear algebra, `libflame`, with support for all major BLAS as well as the most relevant factorization routines for the solution of linear systems. This infrastructure is the basis on which we build our approach for the development of OOC codes.

In our papers, we often start by presenting a prototypical operation, which is then used throughout the paper to illustrate various aspects of the topic at hand. As we have done in a number of other papers that are closely related to the present one [12, 26], we will use the Cholesky factorization as that example. We note that what is different in the current paper is that we focus on the left-looking algorithmic variant for computing the Cholesky factorization. Much of this section can be skipped by those who are very familiar with the FLAME project.

Consider an $n \times n$ Symmetric Positive Definite (SPD) matrix $A$. Its Cholesky factorization is given by $A = LL^T$, where $L$ is the $n \times n$ lower triangular Cholesky factor. In traditional algorithms for this factorization, $L$ overwrites the lower triangular part of $A$ while the strictly upper triangular part remains unmodified. Hereafter, we denote the operation that overwrites $A$ with its Cholesky factor by $A := \{L \backslash A\} = \text{CHOL}(A)$.

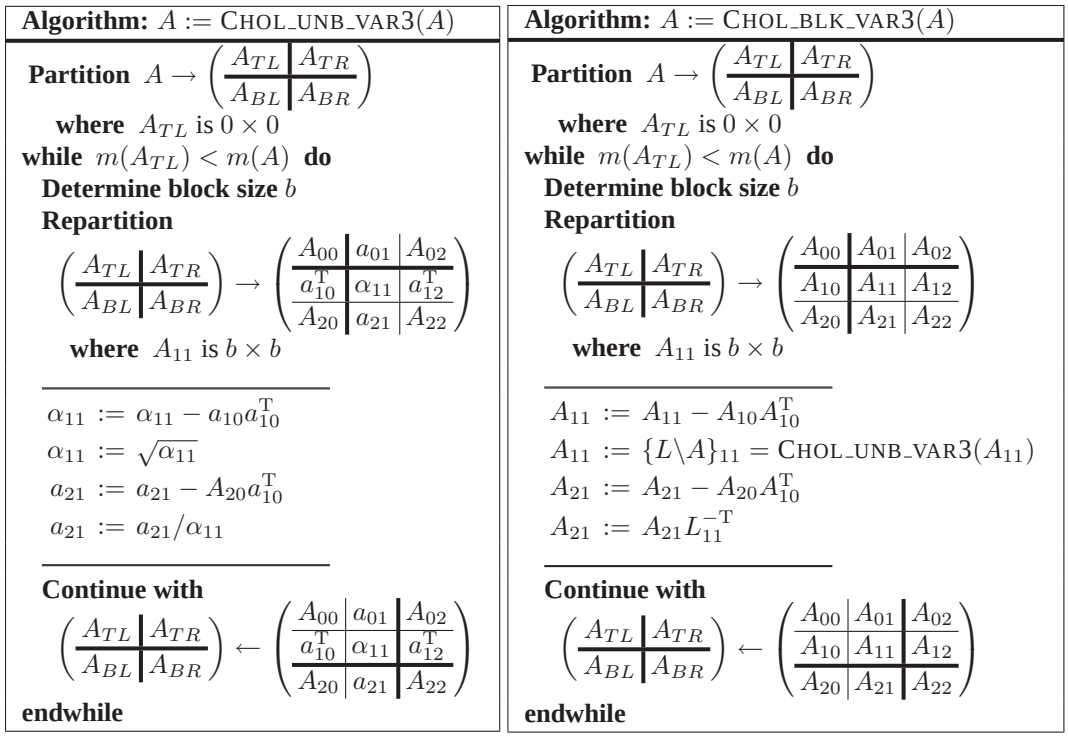


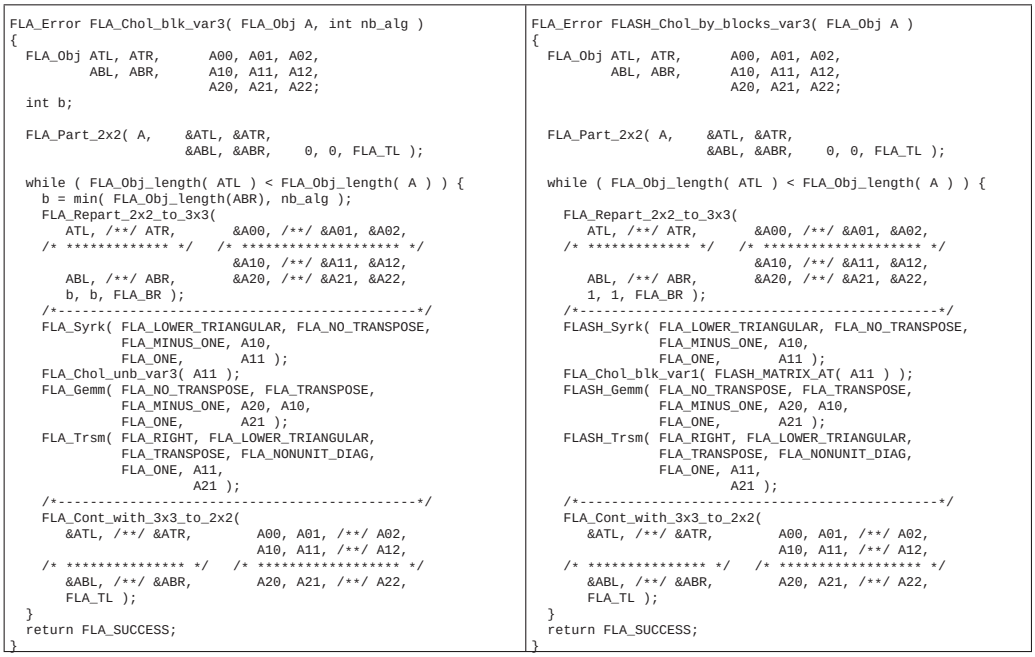

**Figure 1. Unblocked (left) and blocked (right) algorithms for computing the Cholesky factorization (left-looking variant).**

A key element of FLAME is the notation for expressing algorithms much like they are presented on a chalk board [18, 35]. Figure 1 shows unblocked and blocked algorithms for computing the Cholesky factorization using the FLAME notation. There $m(A)$ stands for the number of rows of a matrix $A$. We believe the rest of the notation to be intuitive. The algorithms in the Figure correspond to the "left-looking" algorithmic variant for computing the factorization. It is well-known that this variant requires roughly half the disk I/O when compared with the better-known right-looking variant for the operation.

Using the FLAME/C API for the C programming language, the blocked algorithm in Figure 1 (right) can be transformed into the C code given in Figure 2 (left). Note the close resemblance between algorithm and code: Moving the boundaries of the partitioning imposed on the matrix is performed with routines `FLA_Part_2x2`, `FLA_Repart_...`, and `FLA_Cont_with_...` from the FLAME/C API. The updates during the iteration (loop body) are computed using routines `FLA_Syrk`, `FLA_Gemm`, and `FLA_Trsm`, which are simple wrappers to the analogous BLAS, and routine `FLA_Chol_unb_var3` which corresponds to the FLAME/C unblocked implementation for the algorithm in Figure 1 (left).

With the advent of multi-core processors, the design of *algorithms-by-blocks* [16] for dense linear algebra has regained great interest due to their higher degree of parallelism and better data locality [28]. (In the next section we will show that they are also the key to the OOC implementation of the Cholesky factorization.) Algorithms-by-blocks view matrices as collections of submatrices and express the computation in terms of these submatrix blocks. Algorithms are then written as before, except with scalar operations replaced by operations on the blocks, which now become the unit of computation. We note that one of the first incidences of such algorithms was for OOC dense linear computations and blocks were referred to as tiles [32]. Some refer to algorithms-by-blocks as algorithms-by-tiles or tiled algorithms [11, 10].

A contribution of ours to programmability of algorithms-by-blocks was the recognition that the FLAME/C API could be extended to describe algorithms hierarchically by allowing each element in a matrix to itself be a matrix. We call this very simple extension of FLAME/C the FLASH API [24, 27]. Using the FLASH API an algorithm-by-blocks for the Cholesky factorization is given in Figure 2 (right). The differences between the blocked algorithm and the algorithm-by-blocks in that Figure (left and right, respectively) lie in the dimensions of the partitioning and the routines which are invoked from within the loop body. For the algorithm-by-blocks, the fact that the matrix is indeed a matrix of submatrices, leads to a unit size for the



```
FLA_Error FLA_Chol_blk_var3( 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_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A10,
              FLA_ONE,       A11 );
    FLA_Chol_unb_var3( A11 );
    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE,
              FLA_MINUS_ONE, A20, A10,
              FLA_ONE,       A21 );
    FLA_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 FLASH_Chol_by_blocks_var3( FLA_Obj A )
{
  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 ) );
    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;
}
```

**Figure 2. FLAME/C implementation of the blocked algorithm for the Cholesky factorization (left) and FLASH implementation of the corresponding algorithm-by-blocks.**

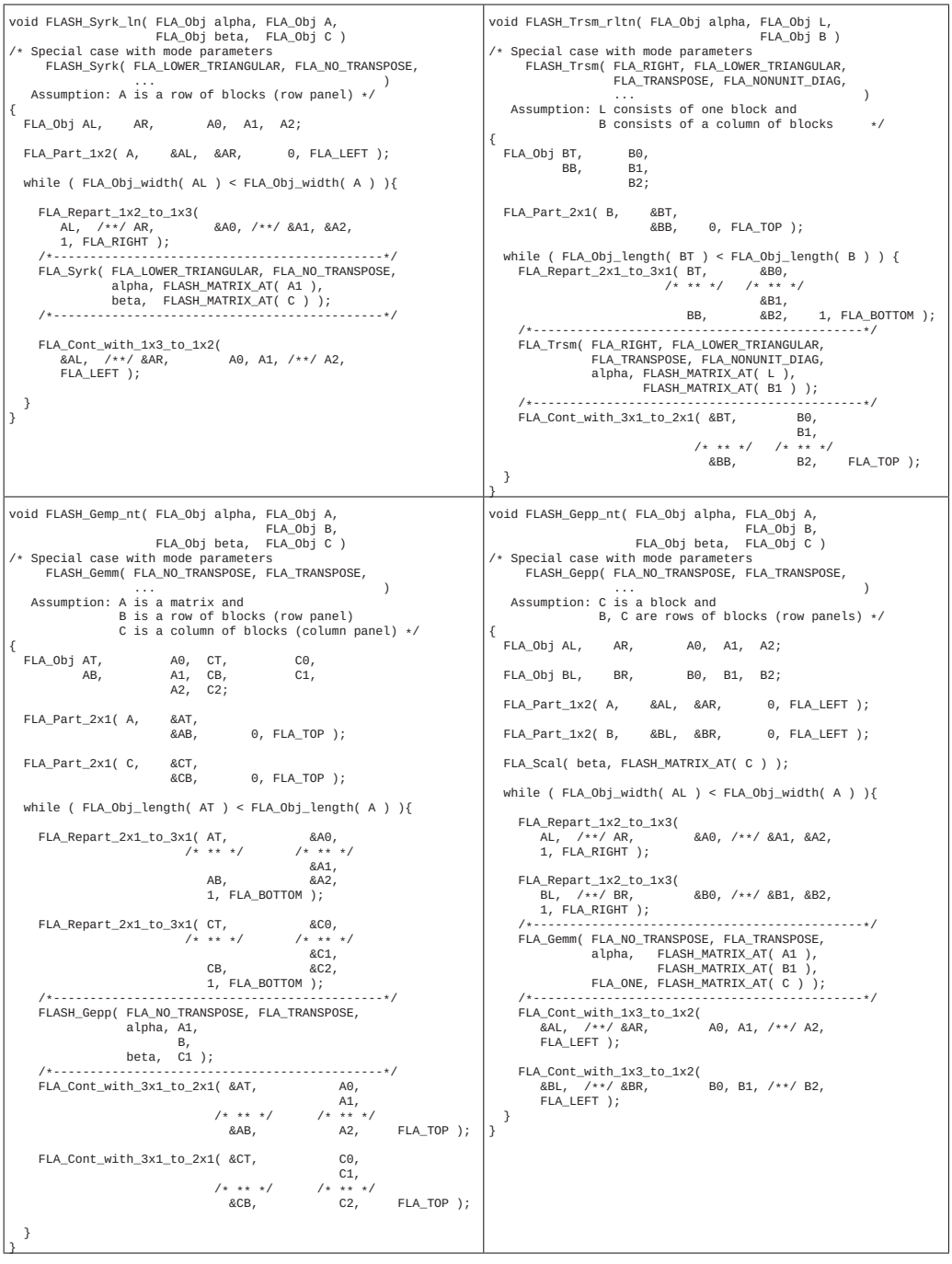
repartitioning operation `FLA_Repart_2x2_to_3x3`. Here, many of the details of the FLASH implementation, including the manipulation of the data structures, have been buried within the FLASH-aware FLAME object definition and the partitioning routines. Abbreviated implementations of algorithm-by-blocks for the building blocks `FLASH_Syrk`, `FLASH_Trsm`, and `FLASH_Gemm` are given in Figure 3. `FLA_Chol_blk_var1` corresponds to the blocked implementation of the right-looking Cholesky factorization, which usually yields higher performance on multi-threaded architectures.

## 3 OOC Implementation

OOC algorithms for dense linear algebra operations traditionally consider a (logical) partitioning of the matrix into submatrices that are stored contiguously on disk. Initially matrices were partitioned into submatrices that were blocks of columns [14, 23][1]. Later it was recognized that this does not scale as matrix sizes become huge (or when memory is relatively small). This is overcome by partitioning the matrix by rows and columns, with the simplest case corresponding to submatrices being square *tiles* (except perhaps for submatrices on the fringe when the matrix size is not an integer multiple of the tile size) [33, 20, 21, 6]. In a nutshell, the reason is that the size of the tile brought into memory can always be kept constant, and therefore the ratio between the computation and I/O overhead can be fixed. Starting from a square partitioning, an OOC algorithm-by-blocks brings a few tiles in-core (usually, to fill a considerable part of the RAM), computes with these, and stores back the results on disk to release space for the data involved in future operations. Optimizing such an OOC implementation becomes a matter of carefully orchestrating the computation so as to bring data into memory for computation in time while (nearly) minimizing the amount of reads and writes (I/O) and/or overlapping I/O and computation. It is particularly the overlapping of I/O with computation (so-called double buffering) that has negatively affected programmability, turning otherwise manageable code into spaghetti code.

In the remainder of this section we present a series of OOC algorithms that start with a basic implementation and culminate in an advanced one that manages all I/O via a run-time system that hides details from the library developer.

[1] Many practical implementations still use this partitioning, especially on clusters with very large memories.



```
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_Gemp_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 and
               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
               B, C are rows of blocks (row panels) */
{
  FLA_Obj AL,    AR,       A0,  A1,  A2;

  FLA_Obj 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 );
    /*-----------------------------------------------*/
    FLA_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 kernels appearing in the FLASH implementation of algorithm-by-blocks for the Cholesky factorization.**

### 3.1 A basic OOC algorithm

An OOC algorithm-by-tiles for the Cholesky factorization is directly obtained from the algorithm-by-blocks in the previous section by just considering the tile to be the unit of computation: The routines in Figures 2 (left) together with those in Figure 3 *are* the OOC implementation.

Given a SPD matrix, created on disk as an OOC matrix of tiles (of dimension $t \times t$), a direct OOC implementation of the Cholesky factorization can be easily obtained from the algorithm-by-tiles by inserting calls to a routine `FLAOOC_Copy` to bring the necessary data into auxiliary workspaces in-core just before the calls to `FLA_Chol_blk_var1`, `FLA_Syrk`, `FLA_Trsm`, `FLA_Gemm`; after the operations, calls to `FLAOOC_Copy` could be inserted to store the results back to disk.

Unfortunately, low performance can be expected from this implementation as, e.g., there is no overlap between I/O and computation. In the next two subsections we describe techniques and tools to improve performance and programmability, including a run-time system that handles I/O transparent to the programmer. As a result, the code does not change: the different schemes that we describe simply change the policy that the run-time system uses to move tiles to and from disk.

### 3.2 Software cache

The first technique to reduce the amount of I/O is to implement a logical cache of tiles that are in-core. The idea is that, every time an operation is to proceed, a run-time inspects the software cache to check whether the tiles the operation involves are already present in-core (cache hit). Thus, actual data transfers only occur for cache misses. A least-recently-used (LRU) replacement policy decides which tile is moved back to disk in case there is no place left in the cache to read a new tile. This is also handled by the run-time.

This simple run-time system handles all I/O transparently to the user, improving the programmability as no explicit I/O calls need to be inserted in the OOC codes. Depending on cache hit rate, it can also improve performance by reducing the number of transfers between RAM and disk. However, it does not overlap I/O with computation yet.

### 3.3 Overlap I/O and computation

The run-time knows what tiles are present in-core (i.e., in the software cache) and therefore can exploit some level of temporal data locality. We next propose going one step further and looking ahead into the future. The domain-specific feature that facilitates the required fortune-telling is that, for linear algebra codes, the operations that will be executed in the future can be known in advance at little cost.

The idea is to perform an initial execution of the code, generating a list of tasks (operations on tiles) to be eventually executed (this is the extra cost we have to pay). Generating a list of tasks for dense linear algebra codes has been used earlier to expose a higher the degree of parallelism at run-time for multi-core processors (see, e.g., [28]). The difference here is that we are proposing to use it with a different goal, namely reducing the amount of data transfers and overlapping computation with I/O for OOC algorithms, in effect prefetching with perfect knowledge.

Let us elaborate the description of this sophisticated run-time system. The codes in Figures 2 (right) and 3, are symbolically executed to generate a list of tasks (*pending list*). Each time a call to routines `FLA_Chol_blk_var1`, `FLA_Syrk`, `FLA_Trsm`, or `FLA_Gemm` is encountered, the run-time simply creates an entry in the list with data to identify the given operation (e.g., operation name and parameters). The order in which the tasks appear in the list together with the directionality of the operands (input or output) defines the order and direction in which blocks will be transferred between memory and disk. Therefore, the future is known in advance!

The real execution can now begin. A single thread, known as the *scout* or *prefect* thread, inspects the *pending list* in (FIFO) order. For each entry of the list, provided there are enough empty (tile) slots in the software cache, the scout thread brings the necessary tiles into the RAM, moving the entry into a second list which contains the tasks which are ready for execution (*ready list*). A second thread, the *worker*, runs over the ready list executing tasks as they are encountered in order. Now, as all data for the computations that are performed by the worker thread are guaranteed to be in-core, we can employ an in-core library for these operations (to be addressed in the next subsection). When a task is completed, the corresponding entry is removed from the ready list, and any tile used within it, which is not used by any other task in the ready list, is marked as candidate for removal from the cache. When new space needs to be allocated in the software cache, the scout thread moves marked tiles back to disk, if they correspond to data that was modified, or simply overwrites them with new data otherwise. When there are no candidates for removal, the scout thread blocks and waits until more tasks are completed.

Probably the most important feature of this approach is how it supports programmability. No change is needed to the algorithm-by-tiles routines. The run-time system is in charge of all data transfers and automatically overlaps I/O with computation. The extra cost for this, creating and managing a couple of lists, is more than paid back by the benefits of reducing idle times due to I/O.

## 4 Parallel In-Core Kernels for Multi-core Processors

In our approach, the worker thread is in charge of computing the operations on tiles which have been already brought in-core by the scout thread. Thus, the types of operations that the worker will encounter are symmetric rank-$k$ updates, triangular system solves, matrix-matrix products, and the computation of the in-core Cholesky factorization of the diagonal tiles (invocations in Figure 3 to `FLA_Syrk`, `FLA_Trsm`, `FLA_Gemm`, and `FLA_Chol_blk_var1`, respectively).

### 4.1 Parallel execution on a general-purpose multi-core processor

Let us consider first that the target architecture is a (general-purpose) processor with several cores (or any other shared-memory platform with multiple processors). For these architectures there exist highly tuned multithreaded implementations of the former three operations, e.g. as part of Intel's MKL library or the GotoBLAS, which efficiently exploit the hardware parallelism. For example, when kernel `_syrk` from MKL is invoked from within routine `FLA_Syrk` to compute a symmetric rank$-k$ update, multiple threads (as many as the user requests) are spawn to compute the operation in parallel, using one core per thread. (Note that in case computation is overlapped with I/O, in our OOC algorithms these threads will run concurrently with the scout thread.) The parallel execution of these three BLAS operations is therefore transparent to the OOC programmer, which only observes a more reduced execution time.

MKL also includes a multithreaded version of the Cholesky factorization which can, in principle, be used to factorize the diagonal tiles using the multiple cores/processors of the architecture. However, for an operation with complex dependencies like this, it may be more efficient to employ a parallelization approach restricted only by the data dependencies (data-flow parallelism). In particular, this second alternative employs the SuperMatrix dynamic scheduling mechanism [27], developed as part of the FLAME project, to improve the scalability of the operation: a scheduling run-time system, different from the one that deals with OOC data transfers and overlap described in the previous section, is in charge of the parallel factorization of the diagonal tiles. In this case, when `FLA_Chol_blk_var1` is invoked, the scheduling run-time inspects the code for this routine, detecting data dependencies among the blocks of the tile, and scheduling for execution those operations (tasks) which have all its dependencies fulfilled. The result is a data-flow parallel execution. Details on dynamic out-of-order scheduling in the context of a parallel execution on multi-core processors can be consulted, e.g., in [28]. In our experiments, we will evaluate the performance of both alternatives: MKL and dynamic scheduling for the factorization of the diagonal tiles. We will refer to the second one as "data-flow" in the experiments.

### 4.2 Parallel execution on a graphics processor

Current workstations include a general-purpose processor (possibly with multiple cores) and usually also a GPU with its own (device) memory. Provided the tile size is large enough and a tuned kernel exists to compute the particular operation on the GPU, the time to transfer data between main (or host) memory and device memory can be more than paid back by the potential of current GPU. For NVIDIA graphics processors, CUBLAS [25, 5] provides an efficient implementation of BLAS-level operations like those needed in the Cholesky factorization, among others. The in-core computation of the Cholesky factorization itself is not provided in NVIDIA libraries but can be easily implemented using the kernels in CUBLAS [4, 36].

Thus, using CUBLAS, once the data is in-core the symmetric rank-$t$ updates, matrix-matrix products, and triangular system solves on tiles are performed in our codes in the GPU. Two different alternatives are explored for the computation of the Cholesky factorization of a (diagonal) tile: a pure CPU computation and a hybrid one, where computation is shared between CPU and GPU. In the pure CPU computation all cores of the general-purpose processor collaborate in the parallel computation of the factorization of the tile using, e.g., the multi-threaded implementation of this operation in MKL. Assume for the hybrid computation that the tile initially resides in the device memory. A blocked right-looking algorithm with block size $b$ is used to compute this factorization as follows: the Cholesky factorization of the diagonal $b \times b$ blocks is computed by the CPU. To do so, the block is initially transferred to host memory, factorized there, and the result is put back into device memory. All other updates of the tile are performed then on the GPU. Once the tile is completely factorized, the result is brought back from the device memory into the corresponding tile of the software cache in host memory.

## 5 Experimental Results

All experiments in this section were performed using MKL 10.0.1, CUBLAS 2.0, and single precision. Performance is measured in terms of GFLOPS (that is, billions of floating-point arithmetic operations –flops– per second), with the usual count of $n^3/3$ flops for the Cholesky factorization. The OOC implementations correspond to the (left-looking) algorithm-by-tiles `FLASH_Chol_by_blocks_var3` in Figure 2 (left). Unless otherwise stated, the enhancements described for the OOC variants are incremental so that a variant includes a new strategy plus those of all previous ones. Several executions were performed to tune the tile size; only the results corresponding to the best case are shown.

### 5.1 Results on a general-purpose multi-core processor

The target architecture for this first experiment is a workstation with two Intel Xeon QuadCore E5405 processors (8 cores) at 2.0 GHz with 8 GBytes of DDR2 RAM. The Intel 5400 chipset provides an I/O interface with a peak bandwidth of 1.5 Gbits/second. The disk is a SATA-I with a total capacity of 160 Gbytes.

Figure 4 reports the performance of several (in-core and OOC) routines for the Cholesky factorization:

**In-core MKL:** The (in-core) implementation of the Cholesky factorization in MKL 10.0.1.

**In-core data-flow:** Our (in-core) algorithm-by-blocks with dynamic scheduling described in [28].

**OOC Basic:** Basic OOC implementation as described in Subsection 3.1.

**OOC Cache:** OOC implementation with a software cache in place to reduce the number of I/O transfers (see Subsection 3.2).

**OOC Reordered + data-flow:** Reordered operations to access tiles following a snake-like pattern to improve locality in the access to the software cache. The factorization of the diagonal tiles is addressed by using the SuperMatrix dynamic scheduling run-time described in [28].

**OOC Overlap I/O:** Use of a run-time with scout and worker threads to overlap computation and I/O, and manage the software cache transparently to the user (see Subsection 3.3).

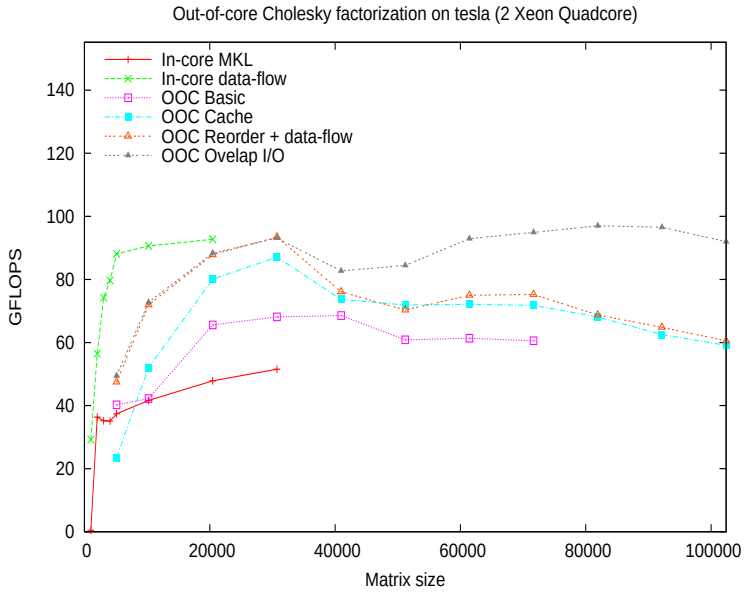



**Figure 4. Performance of the Cholesky factorization codes on a multi-core processor.**

The results in the figure show a practical peak performance for the in-core Cholesky factorization (based on the algorithm-by-blocks, AB) that is slightly over 90 GFLOPS. Combining all the OOC techniques mentioned above, yields a performance that is basically similar to that of the in-core algorithm.

The following table reports the execution time required to compute the Cholesky factorization using the **OOC Overlap I/O** variant and the amount of memory that is needed to store the full dense matrix:

| Matrix size | Time | MBytes |
|---|---|---|
| 10,240 | 4.9sec | 400 |
| 51,200 | 8min 49.9sec | 10,000 |
| 102,400 | 1h 4min 52.0sec | 40,000 |

## 5.2 Results on a multi-core processor+ a graphics processor

We next employ an AMD Phenom 9550 QuadCore at 2.2 GHz with 4 GBytes of DDR2 RAM and 4×512 Kbytes of L2 cache. The chipset provides a I/O interface with a peak bandwidth of 3 Gbits/second. The system has two SATA-II disks (Seagate ST3160815AS, 7200 r.p.m.) with a total capacity of 2×160 Gbytes. The graphics processor is an NVIDIA Geforce 9800 GX2, equipped with 128 cores.

Figure 5 reports the performance of several (in-core and OOC) routines for the Cholesky factorization using the GPU of the system. The timings for the "in-core" results include the cost of transferring data between host and device memories.

**In-core hybrid:** Blocked (in-core) implementation of the Cholesky factorization, with diagonal blocks being factorized in the processor using MKL and all remaining updates in the GPU (see Subsection 4.2).

**OOC Basic:** Basic OOC implementation as described in Subsection 3.1.

**OOC Cache:** Simple run-time that includes a software cache (see Subsection 3.2).

**OOC Hybrid:** Computation of the Cholesky factorization of diagonal tiles being shared between CPU and GPU (see end of Subsection 4.2).

**OOC Overlap I/O:** Elaborated run-time that handles the software cache and overlaps computation and I/O (see Subsection 3.3).

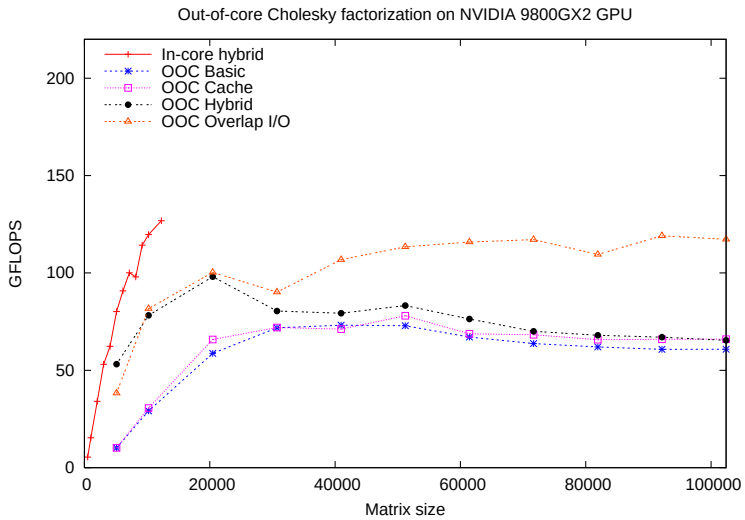



**Figure 5. Performance of the Cholesky factorization codes on a multi-core processor equiped with a GPU.**

The results in the figure show a practical peak performance for the in-core and the best OOC codes that are close to 126 and 117 GFLOPS, respectively. Thus, for this architecture the OOC techniques rival with the in-core algorithm in performance.

Finally, the next table shows the time to compute the Cholesky factorization using the **OOC Overlap I/O** variant and the amount of memory that is needed to store the full dense matrix:

| Matrix size | Time | MBytes |
|---|---|---|
| 10,240 | 4.4sec | 400 |
| 51,200 | 6min 34.6sec | 10,000 |
| 102,400 | 50min 52.3sec | 40,000 |

## 6 An Application Interface

For applications coded in a traditional style, we provide the following prototype API for submitting matrices to disk. We note that, unlike other FLAME/C related routines, these routines expose indexing into a matrix, in order to facilitate how traditional applications may wish to access an otherwise hidden matrix.

Matrices are created on disk with the following call:

```
FLAOOC_Obj_create( FLA_Matrixtype matrixtype, FLA_Datatype datatype,
                   dim_t m, dim_t n, dim_t mt, dim_t nt,
                   char *file_name, FLA_Obj *Aooc );
```

**Purpose:** Create a new object that describes an `m` $\times$ `n` matrix `Aooc`, with entries of type `datatype` (`FLA_INT`, `FLA_REAL`, `FLA_DOUBLE`, etc.), and allocate the associated storage array on disk. The `matrixtype` parameter can be used to roughly half the required space for triangular or symmetric matrices by selecting one from `FLA_LOWER_TRIANGULAR` or `FLA_UPPER_TRIANGULAR`; for full dense matrices, `FLA_DENSE` is to be selected. The matrix is partitioned into `mt` $\times$ `nt` tiles, with the elements in each tile being stored contiguously on disk in column-major order, in a file with name `file_name`.

Arguments `mt` and `nt` are user-defined parameters that must be tuned to optimize performance depending on the problem dimensions, the size of the RAM, and the number of *tiles* (basically square blocks) that must be kept in-core during the execution.

Once created, filling the contents of an OOC object (transferring data from main memory to disk) can be done using the following call:

```
FLAOOC_Copy_submatrix_to_global( FLA_Trans trans, dim_t m, dim_t n,
                                 void *X, dim_t ldim,
                                 dim_t i, dim_t j, FLA_Obj Aooc );
```

**Purpose:** Copy the contents of a conventional column-major matrix `X` with leading dimension `ldim` into the `m` $\times$ `n` submatrix starting at entry (`i`,`j`) of `Aooc`. The `trans` argument may be used to optionally transpose the matrix during the copy.

Assume `FLA_Trans` equals `FLA_NO_TRANSPOSE`. Then, given an `m` $\times$ `n` matrix `X`, in MATLAB notation the previous call is equivalent to

```
Aooc( i:i+m-1, j:j+n-1 ) = X;
```

where `Aooc` is an object with the corresponding data stored on disk. We provide, but do not specify here, a call that can add a multiple of an incore matrix to `Aooc`.

Data transfers in the opposite direction, i.e. from OOC to in-core, can be performed with the following routine:

```
FLAOOC_Copy_global_to_submatrix( FLA_Trans trans, dim_t i, dim_t j, FLA_Obj Aooc,
                                 dim_t m, dim_t n, void *X, dim_t ldim );
```

**Purpose:** Copy the contents of the `m` $\times$ `n` submatrix of `Aooc` whose top-left element is the (`i`,`j`) entry into a conventional column-major matrix `X` with leading dimension `ldim`. The `trans` argument may be used to optionally transpose the matrix during the copy.

Again, an alternative call allows a submatrix from `Aooc` to be added to a contentional matrix `X`.

When no longer needed, a call to `FLAOOC_Obj_free` is required to ensure that all disk space associated with an OOC object is properly released:

```
FLAOOC_Obj_free( FLA_Obj *Aooc );
```

**Purpose:** Release all resources allocated to store data associated with `Aooc` on disk.

Note that these are the routines that a typical application would use to access parts of OOC matrices. The OOC FLAME/C API includes a routine, `FLAOOC_Copy`, which is used to copy tiles to and from disk.

## 7 Concluding Remarks

We have described an approach to easily develop OOC algorithms for dense linear algebra operations. A run-time system in charge of I/O transfers inspects the code before the actual execution begins to bring data from disk before it is needed thus completely hiding I/O latency. As an additional benefit, the run-time system also unburdens the library developer from having to adapt his codes to include routines to explicitly handle the I/O. Thus, all computational routines in `libflame` can be fundamentally transformed into OOC codes without having to change the contents of the library.

Results for an operation like the Cholesky factorization show that the overhead introduced by the run-time is offset by the gains delivered by the overlap of computation and communication (I/O). Using the new run-time, the FLAME code for the left-looking variant of the Cholesky factorization allows to decompose a $100,000 \times 100,000$ matrix on a multi-core platform with 8 cores in slightly more than one hour. A similar workstation equipped with a current GPU can decompose this matrix in approximately 50 minutes.

Future work will include applying the same approach to slab and tiled algorithms for the LU and QR factorization; solving real OOC applications using the resulting codes; and extending the run-time to distributed-memory OOC packages.

### Acknowledgements

The researchers at the Universidad Jaime I were supported by projects CICYT TIN2005-09037-C02-02, TIN2008-06570-C04-01 and FEDER, and P1B-2007-19 P1B-2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. Additional support came from the *J. Tinsley Oden Faculty Fellowship Research Program* of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin.

## References

[1] E. Anderson, Z. Bai, J. Demmel, J. E. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. E. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, 1992.

[2] 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.

[3] M. Baboulin, L. Giraud, S. Gratton, and J. Langou. Parallel tools for solving incremental dense least squares problems. application to space geodesy. Technical Report UT-CS-06-582; TR/PA/06/63, University of Tennessee; CERFACS, 2006. To appear in Journal of Algorithms and Computational Technology, Vol. 3, No 1 (2009).

[4] S. Barrachina, M. Castillo, F. Igual, and R. Mayo adn E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. In Eds. E. Luque et al., editor, *Proceedings of Euro-Par 2008*, number 5168 in LNCS, pages 739–748. Springer-Verlag Berlin Heidelberg, 2008.

[5] 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 *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC'08*, 2008.

[6] Natacha Béreux. Out-of-core implementations of cholesky factorization: Loop-based versus recursive algorithms. *SIAM. J. Matrix Anal. & Appl.*, 30(4):1302–1319, 2008.

[7] 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.

[8] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Trans. Math. Soft.*, 31(1):27–59, March 2005.

[9] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[10] 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.

[11] 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.

[12] 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 2007. ACM.

[13] E. F. D'Azevedo and J. J. Dongarra. The design and implementation of the parallel out-of-core scalapack LU, QR, and Cholesky factorization routines. LAPACK Working Note 118 CS-97-247, University of Tennessee, Knoxville, Jan. 1997.

[14] Eduardo D'Azevedo and Jack Dongarra. The design and implementation of the parallel out-of-core scalapack lu, qr, and cholesky factorization routines. *Concurrency: Practice and Experience*, 12(15):1481–1493, 2000.

[15] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 16(1):1–17, March 1990.

[16] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[17] 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.

[18] 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.

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

[20] Brian C. Gunter, Wesley C. Reiley, and Robert A. van de Geijn. Parallel out-of-core Cholesky and QR factorizations with POOCLAPACK. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, 2001.

[21] 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.

[22] 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.

[23] Ken Klimkowski and Robert van de Geijn. Anatomy of an out-of-core dense linear solver. In *Proceedings of the International Conference on Parallel Processing 1995*, volume III - Algorithms and Applications, pages 29–33, 1995.

[24] Tze Meng Low and Robert van de Geijn. An api for manipulating matrices stored by blocks. Technical Report TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[25] NVIDIA. *CUBLAS Library*. NVIDIA, 2007.

[26] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. In *ACM SIGPLAN 2009 symposium on Principles and practices of parallel programming (PPoPP'09)*, 2009. To appear.

[27] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert van de Geijn, Field Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Transactions on Mathematical Software*. To appear.

[28] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert van de Geijn, Field Van Zee, and Ernie Chan. Programming algorithms-by-blocks for matrix computations to better exploit thread-level parallelism. *ACM Trans. Math. Soft.*, 2008. To appear; available as FLAME Working Note #32 at `http://www.cs.utexas.edu/users/flame/`.

[29] Wesley C. Reiley. Efficient parallel out-of-core implementation of the Cholesky factorization. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Dec. 1999. Undergraduate Honors Thesis.

[30] Wesley C. Reiley and Robert A. van de Geijn. POOCLAPACK: Parallel Out-of-Core Linear Algebra Package. Technical Report CS-TR-99-33, Department of Computer Sciences, The University of Texas at Austin, Nov. 1999.

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

[32] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 1999.

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

[34] Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.

[35] Robert A. van de Geijn and Enrique S. Quintana-Ortí. *The Science of Programming Matrix Computations*. `www.lulu.com`, 2008.

[36] Vasily Volkov and James Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[37] F. G. Van Zee. `libflame`. the complete reference, 2008. In preparation. `http://www.cs.utexas.edu/users/flame`.

[38] Y. Zhang, T. K. Sarkar, R. A. van de Geijn, and M. C. Taylor. Parallel MoM using higher order basis function and PLAPACK in-core and out-of-core solvers for challenging EM simulations. In *IEEE AP-S & USNC/URSI Symposium*, 2008.