Using Graphics Processors to Accelerate the Solution of Out-of-Core Linear Systems[*]

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

Abstract

We investigate the use of graphics processors (GPUs) to accelerate the solution of large-scale linear systems when the problem data is larger than the main memory of the system and storage on disk is employed. Our solution addresses the programmability problem with a combination of the high-level approach in `libflame` *(the FLAME library for dense linear algebra) and a run-time system that handles I/O transparently to the programmer. Results on a desktop computer equipped with an NVIDIA GPU reveal this platform as a cost-effective tool that yields high-performance for solving moderate to large-scale linear algebra problems. The computation of the Cholesky factorization is used to illustrate these techniques.*

1 Introduction

In this paper we target the solution of large dense linear systems on a conventional desktop computer equipped with a graphics processor (GPU). The solution of dense linear systems involving matrices with hundreds of thousands of rows/columns is required, among others, in the estimation of Earth's gravitational field, electromagnetism and acoustics, radar modeling, and molecular dynamics simulations [1, 10, 9, 16, 20]. While the performance boost provided by these hardware accelerators could, in principle, enable the solution of problems this large in reasonable time [4, 3, 19, 13], the data dimension in general exceeds the normal amount of (main) memory in current desktop computers. This can be addressed by changing the mathematical formulation of the underlying problem or the use of secondary memory (e.g., disk). Here we focus on the second alternative, investigating the programmability and performance of high-level out-of-core (OOC) algorithms to solve linear systems on a desktop platform.

The main contributions of this paper are the following:

- There are only a few Open Source libraries for OOC dense linear algebra operations and most of these target distributed-memory platforms [2, 7, 8, 18, 15, 17]. Here we focus on a fundamentally different low-cost architecture: a desktop computer consisting of a multi-core processor (main memory plus disk) and a GPU (device memory) featuring three separate memory spaces.
- We illustrate how the high-level techniques and tools in FLAME accommodate the development of tiled algorithms for dense linear algebra operations. In combination with a run-time system similar to that in SuperMatrix [14], the codes in `libflame` are effortlessly transformed into OOC solvers for dense linear systems.
- We show that, once the problem size becomes large, the performance attained by the OOC implementations rivals that of high-performance algorithms for matrices that fit in-core. Thus, current desktop computers may become a highly cost-effective alternative, making it possible for projects with limited budgets to solve medium to large size problems.

We will illustrate the OOC application programming interface (API), techniques, and tools using the Cholesky factorization of an $n \times n$ symmetric positive definite (SPD)

[*]This research was supported by the CICYT projects TIN2005-09037-C02-02, TIN2008-06570-C04-01, and FEDER, and projects No. P1B-2008-19 P1B-2008-32 of the *Fundación Caixa-Castelló and UJI*.

matrix A, given by $A = LL^T$, where L is the $n \times n$ lower triangular Cholesky factor. This is the first step in the solution of the SPD linear system $Ax = b$, which is followed by the triangular system solves $Ly = b$ and $L^T x = y$. The solution of triangular systems as well as that of general systems (by means, e.g., of the LU or QR factorizations) can be dealt with using ideas similar to those described in the paper.

The paper is structured as follows. In Section 2 we briefly describe a compact API to deal with matrices on disk. A tiled algorithm for the Cholesky factorization and the necessary tiled kernels are reviewed in Section 3. The run-time system that provides both high-performance and easy development of OOC algorithms is presented in Section 4. Finally, Section 5 evaluate the performance on a workstation equipped with a QuadCore AMD processor and an NVIDIA 9800GX2 GPU, and Section 6 summarizes the conclusions and future work.

2 Building an OOC matrix

OOC algorithms for dense linear algebra operations traditionally consider a (logical) partitioning of the matrix on disk into regions, where it is widely recognized that square regions (*tiles*) are preferable over rectangular regions (blocks of columns or rows) [18, 11]. (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.) In this section we describe the use of a simple prototype OOC API [12] to handle matrices *stored-by-tiles* on disk. In this type of storage, elements from the same tile occupy adjacent positions on disk (unless file fragmentation occurs). This layout speeds up reading/writing tiles from/to disk in OOC tiled algorithms.

The fragment of C code in Figure 1 illustrates the use of this API to allocate space on disk for a SPD matrix A composed of $k \times k$ square tiles of dimension $t \times t$ each:

$$A = \begin{pmatrix} B & B/t & 0 & \dots & 0 \\ B/t & B & B/t & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & B/t & B & B/t \\ 0 & \dots & 0 & B/t & B \end{pmatrix},$$

where $B = \text{diag}(1, 2, \dots, t)$ is a $t \times t$ diagonal matrix (tile). (This is a completely artificial problem, meant to illustrate how an application might interface with the library.)

The call to routine FLAOOC_Obj_create in lines 7–9 creates a data structure on disk to hold the entries of the matrix. The parameters of this routine are defined as follows:

```
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 );
```

where datatype specifies the type of the entries (FLA_INT, FLA_FLOAT, FLA_DOUBLE, etc.), m/n define the row/column dimensions of the OOC matrix, mt/nt are the row/column dimensions of the tiles, file_name is the name of the data file on disk, and Aooc is the object for the matrix. Parameter matrixtype can be FLA_DENSE, FLA_LOWER_TRIANGULAR or FLA_UPPER_TRIANGULAR; in the former case, storage is allocated for all tiles of the matrix while, in the other two, space is allocated only for those tiles which contain elements in the lower or upper triangular parts of the matrix. This parameter can thus be employed to roughly half the necessary amount of space on disk when dealing with triangular or symmetric matrices.

The two for loops in the code in Figure 1, in lines 20–29, initialize the contents of the tiles in the lower triangular part of A using an in-core array B with leading dimension ldb. In general, the routine

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

adds to the m×n submatrix of Aooc starting at entry (i,j) the contents of the product alpha·X (trans equals FLA_NO_TRANPOSE) or alpha·X^T (trans equals FLA_TRANSPOSE), where X is a conventional column-major array stored in-core with leading dimension ldim.

The call to routine FLASH_Chol_by_blocks_var3 is then invoked to compute its Cholesky factorization in line 20 and, after ideally processing the results, storage is released in line 22.

The OOC API is complemented with two simple routines to transfer the contents of objects between main memory and disk:

```
FLAOOC_OOC_to_INC(
  FLA_Obj Aooc, FLA_Obj Binc );
FLAOOC_INC_to_OOC(
  FLA_Obj Binc, FLA_Obj Aooc );
```

Note that we distinguish between the FLAOOC_Axpy_submatrix_to_global routine, which is meant to allow applications that explicitly index into arrays to submit (add to) a FLAOOC matrix, and the FLAOOC_X_to_Y routines, which hide details of indexing within the library and are meant for library developers.

```
1   int      k = ...;   /* k x k tiles         */
2   int      t = ...;   /* tile size           */
3   float   *B;         /* = diag(1,2,...,t)   */
4   FLA_Obj  A;         /* OOC matrix          */
5   FLA_Obj  INV_t;     /* Object contains 1/t */
6   /* ... */
7   FLAOOC_Obj_create( FLA_LOWER_TRIANGULAR, FLA_FLOAT, /* Allocate space for A        */
8                      k*t, k*t, t, t,                  /* with k x k tiles of size    */
9                      "File_for_A", &A );              /* t x t each, on disk         */
10  /* Diagonal tiles */
11  for (j=0; j<k; j++)
12    FLAOOC_Axpy_submatrix_to_global( FLA_NO_TRANSPOSE, FLA_ONE,
13                                     t, t,
14                                     B, ldb, j*t, j*t, A );
15  /* Subdiagonal tiles */
16  for (j=0; j<k-1; j++)
17    FLAOOC_Axpy_submatrix_to_global( FLA_NO_TRANSPOSE, INV_t,
18                                     t, t,
19                                     B, ldb, (j+1)*t, j*t, A );
20  FLA_Chol_by_tiles_var3( A ); /* OOC Cholesky factorization */
21  /* ... Use the result */
22  FLAOOC_Obj_free( &A );       /* Release space for A */
```

Figure 1. Fragment of code that allocates a tiled OOC matrix (lines 7-9), initializes its contents (lines 10–19), computes the Cholesky factorization (line 20), and finally releases the space (line 22).

3 Out-of-Core Cholesky Factorization

3.1 FLAME algorithms

Among other tools, FLAME encompasses a notation to describe (dense) linear algebra algorithms at a high level of abstraction and APIs to code these algorithms [5, 6]. Figure 2 (left) shows a tiled (left-looking) algorithm for the Cholesky factorization expressed using the FLAME notation. There, $m(A)/n(A)$ stand for the number of rows/columns of a matrix A, and $A_{11} := \{L\backslash A\}_{11} = \text{CHOL}(A_{11})$ denotes that the lower triangular part of A_{11} is overwritten by the Cholesky factor of the block while the strictly upper triangular part of the matrix remains unmodified. (We follow the traditional implementations for this factorization and overwrite the lower triangular part of A with the Cholesky factor L.) We believe the rest of the notation is intuitive [6].

To illustrate the factorization process performed by the algorithm in Figure 2 (left), consider a matrix of $t \times t$ tiles

$$A = \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix},$$

constructed using the routines in the OOC API described in Section 2. Then, the loop in the algorithm iterates four times, and the repartitioning operation at the the beginning of each iteration exposes the following macrotiles:

Repartition

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

1st iteration

$$\left(\begin{array}{|c|ccc} \hline \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right),$$

2nd iteration

$$\left(\begin{array}{c|c|cc} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \hline \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right),$$

3rd iteration

$$\left(\begin{array}{cc|c|c} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \hline \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{array}\right),$$

4th iteration

$$\left(\begin{array}{ccc|c|} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} & \bar{A}_{03} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} & \bar{A}_{13} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \bar{A}_{23} \\ \hline \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \\ \hline \end{array}\right).$$

Thus, e.g., the following operations are computed during

```
Algorithm: $A := \text{CHOL\_BY\_TILES\_VAR3}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
    where $A_{TL}$ is empty
while $m(A_{TL}) < m(A)$ do
    Repartition
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
        where $A_{11}$ is a tile
    ----------------------------------------
    $A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}}$
    $A_{11} := \{L \backslash A\}_{11} = \text{CHOL}(A_{11})$
    $A_{21} := A_{21} - A_{20}A_{10}^{\mathrm{T}}$
    $A_{21} := A_{21}L_{11}^{-\mathrm{T}}$
    ----------------------------------------
    Continue with
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & A_{01} & A_{02} \\ \hline A_{10} & A_{11} & A_{12} \\ \hline A_{20} & A_{21} & A_{22} \end{array}\right)$
endwhile
```

```
Algorithm: $C := \text{SYRK\_BY\_TILES\_VAR3}(A, C)$
Partition $A \rightarrow \left( A_L \,|\, A_R \right)$
    where $A_L$ is empty
while $n(A_L) < n(A)$ do
    Repartition
    $\left( A_L \,|\, A_R \right) \rightarrow \left( A_0 \,|\, A_1 \,|\, A_2 \right)$
        where $A_1$ is a tile
    ----------------------------------------
    $C := C - A_1 A_1^{\mathrm{T}}$
    ----------------------------------------
    Continue with
    $\left( A_L \,|\, A_R \right) \leftarrow \left( A_0 \,|\, A_1 \,|\, A_2 \right)$
endwhile
```

Figure 2. Tiled algorithms for computing the Cholesky factorization (left) and the corresponding symmetric rank-t update (right).

the third iteration of the algorithm:

$$\begin{array}{l} A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}} \equiv \\ \quad \bar{A}_{22} := \bar{A}_{22} - \left(\bar{A}_{20}\,\bar{A}_{21}\right)\left(\bar{A}_{20}\,\bar{A}_{21}\right)^{\mathrm{T}}, \end{array} \tag{1}$$

$$\begin{array}{l} A_{11} := \{L\backslash A\}_{11} = \text{CHOL}(A_{11}) \equiv \\ \quad \bar{A}_{22} := \{L\backslash \bar{A}\}_{22} = \text{CHOL}(\bar{A}_{22}), \end{array} \tag{2}$$

$$\begin{array}{l} A_{21} := A_{21} - A_{20}A_{10}^{\mathrm{T}} \equiv \\ \quad \bar{A}_{32} := \bar{A}_{32} - \left(\bar{A}_{30}\,\bar{A}_{31}\right)\left(\bar{A}_{20}\,\bar{A}_{21}\right)^{\mathrm{T}}, \end{array} \tag{3}$$

$$A_{21} := A_{21}L_{11}^{-\mathrm{T}} \equiv \bar{A}_{32}\bar{L}_{22}^{-\mathrm{T}}. \tag{4}$$

Consider next a symmetric rank-t update of the form $C := C - AA^{\mathrm{T}}$ where C is a tile while A consists of a row of tiles. A tiled algorithm for this operation is given in Figure 2 (right). When used to compute the symmetric rank-t update in (1), the loop body of the algorithm is executed twice, with $C = \bar{A}_{11}$ and the following repartitionings being imposed on A at the beginning of each iteration:

Repartition

$$\left(A_L \,\Big|\, A_R \right) \rightarrow \left(A_0 \,\Big|\, A_1 \,\big|\, A_2 \right) \equiv$$

$$\underset{\text{1st iteration}}{\left(\Big| \; \bar{A}_{20} \;\big|\; \bar{A}_{21} \right)}, \quad \underset{\text{2nd iteration}}{\left(\bar{A}_{20} \;\Big|\; \bar{A}_{21} \;\big| \right)}.$$

Thus, $\bar{A}_{11} := \bar{A}_{11} - \bar{A}_{20}\bar{A}_{20}^{\mathrm{T}}$ is computed in the first iteration and $\bar{A}_{11} := \bar{A}_{11} - \bar{A}_{21}\bar{A}_{21}^{\mathrm{T}}$ in the second one.

Similar tiled algorithms can be proposed using this notation for the matrix-matrix product in (3) and the triangular system solve in (4).

3.2 FLASH codes

FLAME/C and FLASH are two APIs which allow easy translation of tiled algorithms to C code. In particular, Figure 3 (left) shows the FLASH routine corresponding to the algorithm in Figure 2 (left). We note the close resemblance between algorithm and code; e.g., moving the boundaries of the partitioning imposed on the matrix is performed with routines `FLA_Part_2x2`, `FLA_Repart_2x2_to_3x3`, and `FLA_Cont_with_3x3_to_2x2` from the FLAME/C API. The updates of the matrix are computed with routines `FLASH_Syrk` (symmetric rank-t update $A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}}$), `FLA_Chol_blk_var1` (Cholesky factorization of A_{11}), `FLASH_Gemm` (matrix-matrix product $A_{21} := A_{21} - A_{20}A_{10}^{\mathrm{T}}$), and `FLASH_Trsm` (triangular system solve $A_{21} := A_{21}L_{11}^{-\mathrm{T}}$).

Figure 3 (right) offers code for the symmetric rank-t update that appears during the Cholesky factorization (see also the algorithm in Figure 2 (right)). Routine `FLA_Syrk` in the symmetric rank-t update is a wrapper to corresponding routine in BLAS. Routines `FLASH_Gemm` and `FLASH_Trsm` present analogous implementations, with the operations being decomposed into calls to `FLA_Gemm` and `FLA_Trsm`, which are simple wrappers to the BLAS kernels for the matrix-matrix product and the triangular system solve, respectively.

`FLA_Chol_blk_var1` on the other hand operates on a single tile and corresponds in our implementations to an (in-core) right-looking blocked variant of the Cholesky factorization.

3.3 OOC codes

A (traditional) OOC implementation can be easily obtained from the tiled codes in Figure 3 by simply including the appropriate calls to the routines `FLA_OOC_to_INC` and `FLA_INC_to_OOC` from the OOC API, to explicitly indicate transfers between disk and main memory. For example, an OOC code could be obtained for routine `FLASH_Syrk_ln` by inserting a call to bring the contents of object `C` in-core before the execution of the loop commences, a second call to retrieve the contents of `A1` in-core at the beginning of each iteration, and a final call to set back the result in the in-core copy of `C` to disk once the loop is completed.

We will not pursue this approach further since one of our objectives is to obtain OOC tiled algorithms from a high-level library like `libflame` *without having to modify a single line of the code*.

4 Run-Time System for OOC Matrix Operations

Tiled OOC algorithms are frequently classified as one-tile, two-tile, three-tile,… depending on the (largest) number of these which are kept in-core during the computation. These algorithms try to maximize the size of the tiles in RAM by reducing their number. Our approach is fundamentally different: we will keep a large number of moderately-sized tiles with a two-fold purpose. First, increase the reuse of data in-core by implementing a software cache transparent to the programmer. Second, overlap computation with I/O transfers from disk, also transparently to the programmer, and with no use of asynchronous I/O routines. Therefore our approach aims at *enhancing both performance and programmability* of the codes. As a result, we will show that no substantial changes are required to the codes in library `libflame` to accommodate for OOC computations. The key to this is a *run-time system* that orchestrates computations and I/O transfers.

4.1 Software cache

The first task of our run-time system is to implement a software cache from which the library user can benefit, but one that is transparent to the library programmer at the same time. To achieve this goal, when the user invokes, e.g, a routine from library `libflame` to compute the Cholesky factorization of an OOC matrix, it is our run-time system that takes control over the task. In its basic version (one without overlapping of computation and I/O), the run-time system executes the code symbolically. Each time a call to a FLAME wrapper is encountered (`FLA_Syrk`, `FLA_Gemm`, `FLA_Trsm` or `FLA_Chol_blk_var1`), the run-time identifies the tiles involved in the operation, checks whether the data (tiles) are already in the cache (in-core), and initiates the transfer from disk in case they are not. Once all tiles for a given task are available in-core, the run-time invokes the in-core routine that performs the actual computation. Tiles remain in the cache till space is needed to bring new data. A simple LRU policy defines which tile is moved back to disk, a task which is also carried out by the run-time.

4.2 Overlapping computation and I/O

The software cache exploits temporal data locality to reduce the number of data transfers and improve the performance. We next describe an extension of the run-time which allows overlapped computation and communication, increasing further the performance, without the burden of having to employ asynchronous I/O in the codes.

Let us illustrate how we can do so. Given the codes in Figure 3, the run-time symbolically executes these to generate a list of pending tasks (*pending list*). Each time a call to routines `FLA_Syrk`, `FLA_Gemm`, `FLA_Trsm` or `FLA_Chol_blk_var1` is encountered, the run-time simply creates an entry in the list with data to identify the corresponding operation (e.g., operation code and parameters). The order in which the tasks appear in the list together with the directionality of the operands (input or output) define the order and direction in which blocks will be transferred between in-core and OOC spaces. Therefore, which tiles will be needed "in the future", and in which order, is known before the actual computations commence. Ensuring that tiles are in-core before they are required is thus easily achieved. (Symbolic execution has been also successfully employed to exploit the parallelism of dense linear algebra codes in multicore processors; see, e.g., [14].)

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 that 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 will performed by the worker thread are guaranteed to be in-core, we can employ an in-core library for these operations (more on this 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

```
FLA_Error FLA_Chol_by_tiles_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;
}
```

```
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 );

  }
}
```

Figure 3. FLASH routines for computing the Cholesky factorization (left) and the corresponding symmetric rank-t update (right).

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 overwrites them with new data otherwise. When there are no candidates for removal, the scout thread blocks waiting till more tasks are completed.

Probably the most important feature of this approach is the programmability. No change is needed to the user's codes. The run-time system is in charge of all data transfers and overlapping 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 minimizing idle times due to I/O.

4.3 Computation on the GPU

Once the tiles for a given operation are in-core, it is time to decide where to compute it. Current workstations usually include a general-purpose processor (possibly with multiple cores) as well as 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 RAM (or host memory) and device memory can be more than paid back by the potential performance of the GPU. This is our case: NVIDIA provides CUBLAS [4], an efficient implementation of BLAS-level operations like the symmetric rank-k update, the matrix-matrix product, and the triangular system solve, among others. The computation of the Cholesky factorization (of a tile) is not provided in NVIDIA libraries but can be easily implemented using the kernels in CUBLAS [3, 19].

In our OOC codes, symmetric rank-t updates, matrix-matrix products, and triangular system solves on tiles are performed completely by the GPU. Two different alternatives are explored for the computation of the Cholesky factorization: a pure CPU computation and a hybrid one, where computation is shared between CPU and GPU. In particular, for the hybrid computation, assume the matrix 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 on a workstation with an AMD Phenom 9550 QuadCore at 2.2 GHz with 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 (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. MKL 10.0.1.014, CUBLAS 2.0, and single precision were employed in the experiments. Performance is measured in terms of GFLOPS (that is, billions of arithmetic floating-point operations –flops– per second), with the usual count of $n^3/3$ flops for the Cholesky factorization.

Figure 4 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. All OOC implementations correspond to the tiled OOC routine `FLASH_Chol_by_blocks_var3` in Figure 3 (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.

In-core hybrid: Blocked (in-core) implementation of the Cholesky factorization, with diagonal blocks being factorized in the processor and all remaining updates in the GPU (see subsection 4.3).

OOC traditional: Explicit I/O routines to transfer tiles inserted in the code.

OOC cache: Simple run-time that implements a software cache (see subsection 4.1).

OOC hybrid: Computation of the Cholesky factorization of diagonal tiles being shared between CPU and GPU (see end of subsection 4.3.

OOC I/O overlap: Elaborated run-time that handles the software cache and overlaps computation and I/O (see subsection 4.2).

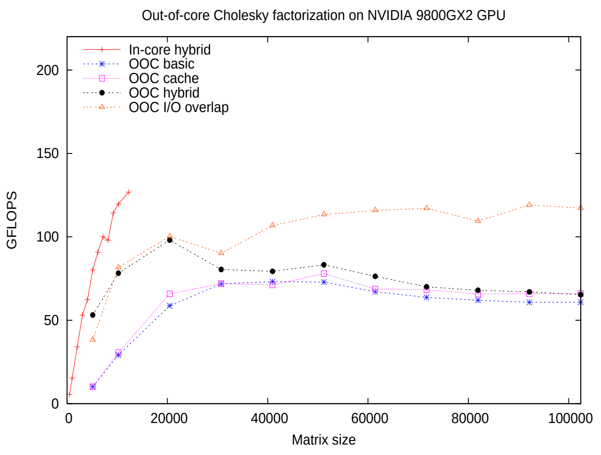

Figure 4. Performance of the Cholesky factorization codes.

The results in the figure show a practical peak performance for the in-core and the best OOC codes that are around 126 and 117 GFLOPS, respectively. Using the new run-time and an unexpensive GPU, the OOC code for the left-looking variant of the Cholesky factorization allows to factorize a matrix of dimension $100,000 \times 100,000$ matrix in slightly over 45 minutes.

6 Concluding Remarks

We have described an efficient approach to solve linear systems which are too large to fit into main memory exploiting the capabilities of the GPU as a hardware accelerator. A run-time system analyzes the code before the actual execution begins, to schedule data transfers from disk in due time, 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 routine calls to explicitly handle the I/O. Following this approach, all computational routines in `libflame` become OOC codes without having to change the contents of the library.

Results for a complex operation like the Cholesky factorization show that the overhead introduced by the run-time is completely compensated by the gains delivered by the overlap of computation and communication (I/O).

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.

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] 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.

[3] 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.

[4] 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 Int. Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC'08*, 2008.

[5] 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 Trans. Math. Softw.*, 31(1):1–26, March 2005.

[6] 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. Softw.*, 31(1):27–59, March 2005.

[7] 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.

[8] 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.

[9] 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.

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

[11] Brian C. Gunter and Robert A. van de Geijn. Parallel out-of-core computation and updating the QR factorization. *ACM Trans. Math. Softw.*, 31(1):60–78, March 2005.

[12] M. Marqués, E. S. Quintana-Ortí, and G. Quintana-Ortí. FLAOOC/C: A high-level API for out-of-core linear algebra operations. Technical Report ICC X-2008, Depto. de Ingenieria y Ciencia de Computadores, Universidad Jaume I, 2008. In preparation.

[13] Gregorio Quintana-Ortí, Francisco D. Igual , Enrique S. Quintana-Ortí, and Robert van de Geijn. Extending the SuperMatrix runtime scheduling system for platforms with multiple hardware accelerators. In *The 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP 2009*, Raleigh, NC, USA, 2009. To appear.

[14] 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. Softw.*, 2009. To appear.

[15] 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.

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

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

[18] 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.

[19] V. Volkov and J.W. Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Dept., University of California, Berkeley, May 2008.

[20] 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.