Informe Técnico ICC 01-05-2008

Out-of-Core Solution of Linear Systems on Graphics Processors

Maribel Castillo, Francisco D. Igual, Rafael Mayo, Rafael Rubio,
Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert van de Geijn

Mayo de 2008

Departamento de Ingeniería y Ciencia de Computadores

Correo electrónico: {`castillo, figual, mayo, gquintan, quintana`}`@icc.uji.es, rvdg@cs.utexas.edu`

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

Out-of-Core Solution of Linear Systems on Graphics Processors

Maribel Castillo[1],
Francisco D. Igual[2],
Rafael Mayo[3],
Rafael Rubio[4],
Gregorio Quintana-Ortí[5],
Enrique S. Quintana-Ortí[6],
Robert van de Geijn[7],

Abstract:

We combine two high-level application programming interfaces to solve large-scale linear systems with the data stored on disk using current graphics processors. The result is a simple yet powerful tool that enables a fast development of object-oriented codes, implemented as MATLAB M-scripts, for linear algebra operations.

The approach enhances the programmability of the solutions in this problem domain while unleashing the high performance of graphics processors. Experimental results are reported from OCTAVE , linked with the implementation of BLAS by NVIDIA, and executed on a G80 processor.

Keywords:

Graphics processors (GPUs), general purpose computing on GPU, linear algebra, BLAS, high performance.

[1] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: castillo@icc.uji.es.
[2] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: figual@icc.uji.es.
[3] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: mayo@icc.uji.es.
[4] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: rubio@icc.uji.es.
[5] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: gquintan@icc.uji.es.
[6] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: quintana@icc.uji.es.
[7] Computer Science Department. The University of Texas at Austin
E-mail: rvdg@cs.utexas.edu.

Resolución Out-of-Core de Sistemas Lineales sobre procesadores gráficos

Maribel Castillo[8],
Francisco D. Igual[9],
Rafael Mayo[10],
Rafael Rubio[11],
Gregorio Quintana-Ortí[12],
Enrique S. Quintana-Ortí[13],
Robert van de Geijn[14],

Resumen:

Se combinan dos APIs de alto nivel para la resolución de sistemas lineales de gran escala con los datos almacenados en disco, mediante el uso de procesadores gráficos de última generación. El resultado es una herramienta simple, aunque de gran potencia, que permite el rápido desarrollo de códigos orientados a objetos, implementados como M-scripts, para operaciones de álgebra lineal.

Nuestra propuesta mejora la facilidad de programación para este tipo de problemas, y a la vez aprovecha el elevado rendimiento de los procesadores gráficos. Los resultados experimentales han sido extraídos de OCTAVE, enlazado con la implementación de BLAS de NVIDIA, y ejecutados sobre un procesador G80.

Palabras clave:

Procesadores gráficos (GPUs), procesamiento de carácter general sobre GPUs, álgebra lineal, BLAS, altas prestaciones.

[8] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: castillo@icc.uji.es.
[9] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: figual@icc.uji.es.
[10] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: mayo@icc.uji.es.
[11] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: rubio@icc.uji.es.
[12] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: gquintan@icc.uji.es.
[13] Departamento de Ingeniería y Ciencia de los Computadores
E-mail: quintana@icc.uji.es.
[14] Computer Science Department. The University of Texas at Austin
E-mail: rvdg@cs.utexas.edu.

Out-of-Core Solution of Linear Systems on Graphics Processors

Maribel Castillo[1], Francisco D. Igual[1], Rafael Mayo[1], Enrique S. Quintana-Ortí[1], Gregorio Quintana-Ortí[1], Rafael Rubio[1], and Robert van de Geijn[2]

[1] Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
{castillo,figual,mayo,quintana,gquintan,rubio}@icc.uji.es
[2] Department of Computer Sciences,
The University of Texas at Austin, Austin, Texas 78712,
rvdg@cs.utexas.edu

Abstract We combine two high-level application programming interfaces to solve large-scale linear systems with the data stored on disk using current graphics processors. The result is a simple yet powerful tool that enables a fast development of object-oriented codes, implemented as MATLAB M-scripts, for linear algebra operations.
The approach enhances the programmability of the solutions in this problem domain while unleashing the high performance of graphics processors. Experimental results are reported from OCTAVE , linked with the implementation of BLAS by NVIDIA, and executed on a G80 processor.

Key words: Out-of-core computing, linear algebra, high-level APIs, user-friendly problem-solution environments, graphics processors (GPUs).

1 Introduction

The OOC solution of dense linear algebra problems arises, among others, in Boundary Element Methods (BEM) for integral equations in electromagnetics and acoustics, and in radial function methods [7,9,10]. When the data structures involved in these problems are too large to fit in memory, the only solution is to rely on disk storage. Although such additional memory can be accessed via virtual memory, careful design of Out-of-Core (OOC) algorithms is generally required to attain high performance.

ScaLAPACK and SOLAR contain parallel routines for the OOC solution of linear systems on distributed-memory parallel systems [14,8]. The POOCLAPACK package [6] offers a more friendly, object-oriented (OO) application programming interface (API) for OOC computing also on this type of platforms.

Our approach departs from the previous OOC software efforts in two crucial aspects, which are the main contributions of this paper: 1) We follow and extend the tools in the FLAME project (`http://www.cs.utexas.edu/users/flame`) with a new high-level OO API for constructing OOC dense linear algebra algorithms from user-friendly environments like MATLAB, OCTAVE, or LABVIEW. We believe such an API offers an important added value of wide appeal to a majority of scientists and engineers, who prefer this class of environments to perform complex analysis, modeling, and simulations; and 2) we target a fundamentally different architecture, namely graphics processor (or GPU), which is rapidly becoming a standard hardware accelerator for certain computing-intensive operations. Our experimental results reveals the remarkable properties of graphics processors to deal with large-scale dense linear algebra problems.

The rest of the paper is structured as follows. In Section 2 we offer a few remarks on the OOC solution of linear systems. Sections 3 and 4 are devoted, respectively, to briefly reviewing the FLAME approach to coding dense linear algebra operations and presenting in more detail the corresponding extension for OOC computing on graphics processors. Experimental results on a NVIDIA graphics processor G80 report the performance of the approach in Section 5 and, finally, the conclusions follow in Section 6.

2 OOC Solution of Linear Systems

In this paper we employ the Cholesky factorization as a representative example of the OOC solution dense linear systems. In this operation, a symmetric positive definite $n \times n$ matrix A is decomposed into the product A = LL^T , where the lower triangular $n \times n$ matrix L is the Cholesky factor of A. (Alternatively, A can be decomposed as $A = U^T U$, with U an $n \times n$ upper triangular matrix.)

In this factorization, roughly $n^3/3$ floating-point arithmetic operations (flops) are performed on n^2 numbers. (In practice, the entries of the Cholesky factor L overwrite the corresponding entries of the lower triangular part of A as they are computed so that only one matrix is involved.) This results in a computationally-bounded operation of wide appeal to current desktop systems, with a deep memory hierarchy, and also to graphics processors, as will be shown later.

OOC algorithms for the Cholesky factorization usually proceed by logically partitioning the matrix stored on disk into $t \times t$ tiles, $\bar{A}_{i,j}$, (or square blocks):

$$A = \begin{pmatrix} \bar{A}_{0,0} & \bar{A}_{0,1} & \dots & \bar{A}_{0,N-1} \\ \bar{A}_{1,0} & \bar{A}_{1,1} & \dots & \bar{A}_{1,N-1} \\ \vdots & \vdots & \ddots & \vdots \\ \bar{A}_{N-1,0} & \bar{A}_{N-1,1} & \dots & \bar{A}_{N-1,N-1} \end{pmatrix},$$

and bringing in-core only those tiles that participate in the current suboperation [11] . (In general, between 1 and 3 tiles are kept in-core at any given stage.)

Procedures for the OOC computation of the LU factorization with partial pivoting and the QR factorization, on the other hand, have traditionally viewed the matrix as partitioned into $n \times s$ *slabs* (column blocks or panels). More recently, tiled algorithms have been proposed for the QR factorization [11] and the LU factorization with incremental pivoting [12,13]. Thus, the use of tiles as the basic building block for the OOC API that is advocated here is general enough to cover the solution of linear systems (and linear least-squares problems).

```
Algorithm: $A := \text{CHOL\_UNB}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
  Repartition
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{T} & \alpha_{11} & a_{12}^{T} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
  where $\alpha_{11}$ is $1 \times 1$

    $a_{10}^{T} := a_{10}^{T}\text{TRIL}(A_{00})^{-T}$
    $\alpha_{11} := \alpha_{11} - a_{10}^{T}a_{10}$
    $\alpha_{11} := \sqrt{\alpha_{11}}$

  Continue with
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \leftarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{T} & \alpha_{11} & a_{12}^{T} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
endwhile
```

```
Algorithm: $A := \text{CHOL\_BLK}(A)$
Partition $A \rightarrow \left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right)$
  where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
  Determine block size $n_b$
  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 $n_b \times n_b$

    $A_{10} := A_{10}\text{TRIL}(A_{00})^{-T}$
    $A_{11} := A_{11} - A_{10}A_{10}^{T}$
    $A_{11} := \text{CHOL\_UNB}(A_{11})$

  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
```

Figure 1. Left-looking unblocked (left) and blocked (right) algorithms for computing the Cholesky factorization.

The OOC solution of the triangular linear systems resulting from the previous factorizations basically implies the same techniques described next while presenting a minor computational cost. Therefore, we do not discuss that final stage further.

3 An OO API to Coding Dense Linear Algebra Algorithms

The FLAME project encompasses a formal methodology for deriving algorithms for dense linear algebra operations, a notation to express these algorithms with a high level of abstraction, and several APIs to transform the algorithms into codes [4]. Key to FLAME is the use of an OO approach, much as in scientific computing projects like PLAPACK [15] or PETSc [1], for both expressing algorithms and codes. This is illustrated in Figure 1, which exposes unblocked and blocked algorithms for the Cholesky factorization of a matrix using the FLAME notation. There, $m(B)$ and `TRIL`(B) denote, respectively, the number of rows and the lower triangular part of B. We believe the rest of notation to be intuitive; for details, visit the FLAME website or consult [5].

Using the FLAME@lab M-script API, the blocked algorithm in Figure 1 (right) is encoded as shown in Figure 2. A comparison of these two figures illustrates how the use of a high-level API allows the code to closely mirror the algorithm, thus hiding implementation details into data objects (blocks or submatrices) and the corresponding functions following an OO approach. In particular, note how FLAME avoids complicated indexing by creating views, new objects or references into an existing matrix or vector, through the partitioning operations.

```
function [ A_out ] = FLA_Chol( A, nb_alg )

  [ ATL, ATR, ...
    ABL, ABR ] = FLA_Part_2x2( A, ...
                               0, 0, 'FLA_TL' );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) )
    b = min( nb_alg, FLA_Obj_length( ABR ) );
    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = FLA_Repart_2x2_to_3x3( ATL, ATR, ...
                                               ABL, ABR, ...
                                               b, b, 'FLA_BR' );

    %------------------------------------------------------------------%
    A10 = A10 / tril( A00 )';                  % A10 := A10 * TRIL(A00)^-T
    A11 = A11 - A10 * A10';                    % A11 := A11 - A10 * A10^T
    A11 = chol(A11)';                          % A11 := Chol( A11 )'
    %------------------------------------------------------------------%

    [ ATL, ATR, ...
      ABL, ABR ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                             A10, A11, A12, ...
                                             A20, A21, A22, ...
                                             'FLA_TL' );
  end
  A_out = ATL;
return
```

Figure 2. FLAME@lab code to compute the Cholesky factorization of a matrix.

4 An OO API to Coding OOC Dense Linear Algebra Algorithms

The need to operate on the GPU with data matrices initially residing on disk and produce results that eventually will be transferred back to disk led us to develop a collection of routines in the OOC API structured in three major groups:

- OOC data structure handling (`FLAOOC_`): Routines in the FLAOOC@lab API [15] to create and destroy matrices-by-tiles on disk, set and retrieve their contents, and track the movement through them during the execution of the tiled OOC algorithm.
- I/O transfer (`FLAGOOC_`): Routines to move data between disk and GPU memory.
- GPU computation (`FLAG_`): Routines in the FLAG@lab API [3] to compute with matrices stored on the GPU memory.

We next describe the major elements of the FLAGOOC@lab API in each one of these groups using the Cholesky factorization as the guiding example.

4.1 OOC data structure handling

The fragment of M-script code in Figure 3 allocates space on disk for an $n \times n$ SPD matrix

$$A = \begin{pmatrix} B & B/t \\ B/t & B \end{pmatrix},$$

with $B = \mathrm{diag}(1, 2, \ldots, t)$ being a diagonal matrix and tile size $t = n/2$ (lines 5-7). Routine `FLAOOC_Obj_create` there allows the user to specify the data type of the entries of the OOC matrix (real or complex), its dimensions (rows/columns) and tile size, and the name to use for the file on disk. We note that the $n \times n$ matrix is never explicitly formed and only resides on disk. In particular, A is an object (or descriptor) for the OOC matrix. The tile size is a user-selected parameter that must be tuned for optimal performance depending on the problem dimensions, the size of the GPU memory, and the number of tiles that must be kept in the GPU during the algorithm. The tile size dictates how the partitioning/repartitioning routines operating on this matrix will proceed.

The contents of the $t \times t$ tiles in the lower triangular part are initialized next with three calls to `FLAOOC_Axpy_matrix_to_object` (lines 9-14); each invokation to this routine adds the contents of the object B, scaled by the first argument, to the part of the OOC matrix A starting at the entry specified by the last two arguments. Routine `FLAGOOC_Chol` (to be illustrated later) is invoked then to compute the Cholesky factorization (line 16) and, after using the result, the space for A on disk is deallocated (line 18).

```
 1  n = ...;                                % Matrix size
 2  t = n/2;                                % Tile size
 3  B = diag([1:t]);                        % Building block
 4
 5  A = FLAOOC_Obj_create( 'FLA_REAL',      % Entries are real numbers
 6                         n, n, t,         % n x n matrix with tile size t
 7                         'File_for_A' );  % File name on disk
 8
 9  FLAOOC_Axpy_matrix_to_object( 1,   B,             % B in A(1:t,1:t)
10                                     A, t,    t );
11
12  FLAOOC_Axpy_matrix_to_object( 1/t, B,             % B/t in A(t+1:n,1:n)
13                                     A, t+1,  1 );
14
15  FLAOOC_Axpy_matrix_to_object( 1,   B,             % B in A(t+1:n,t+1:n)
16                                     A, t+1, t+1 );
17
18  A = FLAGOOC_Cholesky( A );             % Compute the Cholesky factor
19  ...                                    % Somehow use the result
20  FLAOOC_Obj_destroy( A );               % Free storage
```

Figure 3. Computation of the Cholesky factorization of an OOC matrix.

Figure 4 illustrates an OOC algorithm for the Cholesky factorization encoded using the OOC API. Note the similarities with the blocked code in Figure . The partitioning and repartitioning in the OOC code are just indexing operations which do not modify the contents of the matrix. These operations track the initial partitioning of the OOC matrix into $t \times t$ tiles that was defined upon creation of the matrix. Thus, e.g., in the 3×3 repartitioning resulting from `FLAOOC_Repart_2x2_to_3x3`, `A11` is a tile extracted from `ABR`, which is later incorporated into `ATL` in the call to `FLAOOC_Cont_with_3x3_to_2x2`. Thus, the fact that the algorithm operates on matrices partitioned and stored on disk by tiles is mostly transparent to the user. All results and arguments in the partitioning routines are objects or descriptors to the OOC matrix, not matrices themselves.

4.2 I/O transfer

All movement of data between disk and the GPU memory is done using two routines: FLAGOOC_OOC_to_GPU and FLAGOOC_GPU_to_OOC (see, e.g., lines 20 and 25 in Figure 4). The result and arguments of these two routines are objects or descriptors of matrices which reside on disk or GPU memory.

4.3 GPU computation

All the actual operations on the data occur in between the dashed lines of the code; see Figure 4. Routines FLAGOOC_Trsm and FLAGOOC_Syrk employ the FLAGOOC@lab API to solve a triangular linear system and compute a symmetric rank-k update, respectively. Figure 5 provides a special instance of the implementation of the latter. As shown there, at the bottom level routine FLAGOOC_Syrk routine is decomposed into calls to FLAG_Syrk, a routine from the FLAG@lab API. This GPU API covers the functionality of the basic linear algebra subprograms (BLAS). Similarly, FLAGOOC_Trsm is decomposed into calls to the routines in FLAG@lab.

```
 1  function [ A_out ] = FLAGOOC_Chol( A )
 2    [ ATL, ATR, ...
 3      ABL, ABR ] = FLAOOC_Part_2x2( A, ...
 4                                 0, 0, 'FLA_TL' );
 5
 6    while ( FLAOOC_Obj_length( ATL ) < FLAOOC_Obj_length( A ) )
 7      [ A00, A01, A02, ...
 8        A10, A11, A12, ...
 9        A20, A21, A22 ] = FLAOOC_Repart_2x2_to_3x3( ATL, ATR, ...
10                                                  ABL, ABR, ...
11                                                  1, 1, 'FLA_BR' );
12      %------------------------------------------------------------------%
13      A10  = FLAGOOC_Trsm( 'FLA_RIGHT',      'FLA_LOWER_TRIANGULAR',
14                           'FLA_TRANSPOSE',  'FLA_NONUNIT_DIAG',
15                           1, A00,
16                           A10 );                 % A10 := A10 * TRIL(A00)^-1
17      AGPU = FLAGOOC_OOC_to_GPU( A11 );           % Copy A11 from OOC to GPU
18      AGPU = FLAGOOC_Syrk( 'FLA_LOWER_TRIANGULAR', 'FLA_NO_TRANSPOSE',
19                           -1, A10,
20                            1, AINC );            % A11 := A11 - A10 * A10^T
21      AGPU = FLAG_Chol(AGPU);                     % A11 := chol( A11 )
22      FLAGOOC_GPU_to_OOC( AGPU, A11 );            % Copy AGPU from GPU to OOC
23      %------------------------------------------------------------------%
24
25      [ ATL, ATR, ...
26        ABL, ABR ] = FLAOOC_Cont_with_3x3_to_2x2( A00, A01, A02, ...
27                                                A10, A11, A12, ...
28                                                A20, A21, A22, ...
29                                                'FLA_TL' );
30    end
31    A_out = ATL;
32  return
```

Figure 4. FLAGOOC@lab code to compute the Cholesky factorization of an OOC matrix.

A careful inspection of these codes reveals a certain inefficiency in the manner of computing the Cholesky factorization: some of the copies that transfer tiles of A10 to

```
function [ C_out ] = FLAGOOC_Syrk( uplo, trans, alpha, A, beta, C )
% Assumption: A is a row of tiles (row panel) and
% C is a single tile already in the GPU

  C = FLAG_Scal( beta, C );                % C := C + beta * C

  [ AL, AR ] = FLAOOC_Part_1x2( A, ...
                              'FLA_LEFT' );

  while ( FLAOOC_Obj_width( AL ) < FLAOOC_Obj_width( A ) )

   [ A0, A1, A2 ]= FLAOOC_Repart_1x2_to_1x3( AL, AR, ...
                                           1, 'FLA_RIGHT' );

   %----------------------------------------------------------------------%
   AGPU = FLAGOOC_OOC_to_GPU( A1 );         % Copy A1 from OOC to GPU
   C = FLAG_Syrk( uplo, trans,
                  alpha, AGPU,
                  1, C );                  % C := C + alpha * A1 * A1'
   %----------------------------------------------------------------------%

   [ AL, AR ] = FLAOOC_Cont_with_1x3_to_1x2( A0, A1, A2, ...
                                           'FLA_LEFT' );

 end
 C_out = C;
return
```

Figure 5. FLAGOOC@lab code to compute the symmetric rank-k update involved int eh Cholesky factorization of an OOC matrix.

GPU memory during execution of the routine FLAGOOC_Trsm could be, in principle, overlapped with those performed inside FLAGOOC_Syrk. For the sake of simplicity, we obviate this detail in the presentation of the codes. Also, during the execution of these routines, the contents of the strictly upper triangular part of the diagonal $t \times t$ tiles of A is destroyed. Once more, for clarity, we prefer to obviate how this can be fixed in the codes.

5 Experimental Results

In this section we offer some results that illustrate the performance of the OOC routine for the Cholesky factorization in Figure 4. The experiment was conducted using single-precision arithmetic on an NVIDIA GeForce 8800 Ultra (processor G80 running at 575MHz with 768 Mbyte of DDR RAM memory). OCTAVE 2.9.19 linked with the implementation of the CUDA BLAS (CUBLAS 1.1) was employed. Our own tuned codes were built upon the NVIDIA BLAS to solve triangular linear systems (FLAG_Trsm), compute a rank-k update (FLAG_Syrk), and obtain the Cholesky factorization (FLAG_Chol). The CPU of the server for the GeForce board is an Intel Core2Duo with two cores at 1.86 GHz and 1 Gbyte of DDR RAM, running a Linux kernel version 2.6.18.

Figure 6 reports de GFLOPs delivered by the routine. For reference, we also include the GFLOPs attained by MATLAB routine chol, that internally invokes the routine spotrf from the MKL 10.0 implementation of LAPACK on the CPU, using one core of the Intel processor. Similar behavior has been observed when using both cores.

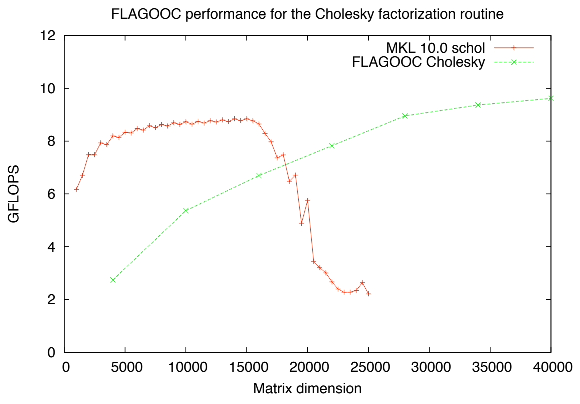

Figure 6. Performance of the Cholesky factorization of an OOC matrix compared with the performance of the MKL implementation of the routine

For the OOC routine, we carry out several experiments to determine the optimal tile size t, but only report the results for the best value. Due to the stream oriented architecture of the GPUs, the best performance results are usually achieved for big data streams. The only limitation in our case is the physical memory limit imposed by the GPU (768 Mbytes in our experimental setup). The observed optimal value for the block size for the GeForce system was $t = 6000$.

The figure shows that, when the matrix dimension exceeds the RAM capacity, the virtual memory system does not handle efficiently the data, and the performance of the routine `chol` decreases rapidly. On the other hand, the OOC routine, while attaining a smaller percentage of peak performance than its MATLAB counterpart (basically due to I/O overhead), maintains its performance for large-scale problems.

In addition to the data transfer bottleneck, there are three main reasons why the peak performance of the FLAGOOC is still relatively low. First, current GPUs only operate in simple precision, while the OCTAVE setup used in our experiments only works in double precision. Though future generations may support double precision arithmetics natively, it is necessary to perform an intermediate conversion of data before transferring it to the video memory, or back to OCTAVE objects. Second, the CUBLAS implementation is not fully optimized [2], specially for routines that are heavily used in our algorithm (TRSM and SYRK). Being a CUBLAS-based implementation, FLAG@lab relies on the performance of the underlying BLAS implementation developed by NVIDIA. Third, there exist some intrinsic limitations in the MATLAB/OCTAVE code that suppose an important overhead for the final performance. In addition to the interpreted nature of the M-script code, there is an extra overhead related to the mechanism used by MATLAB/OCTAVE to manage the invocation of external compiled codes. In our implementation, intensive CUDA/CUBLAS calls are performed, with an important performance penalty related to this limitation.

However, besides the quantitative performance results, that can be solved by using alternative solutions (e.g. C implementations), our prototype implementation throws interesting qualitative results, mainly observed for big matrices, for which MKL, or any other implementation with no specific support for OOC situations, cannot attain high performance.

6 Concluding Remarks

The FLAGOOC@lab API provides an easy-to-use tool to develop codes for dense linear algebra operations with matrices stored on disk and execute them on a graphics processor. The simplicity of the solution comes from the adoption of the FLAME notation and tools (including the FLASH API for hierarchically storing matrices by blocks), while much of its appeal is in the user-friendly access to the FLAGOOC@lab API via environments like MATLAB/OCTAVE. Combined with an implementation of BLAS for a graphics processor, the proposed interface allows an efficient solution of large-scale OOC dense linear algebra problems on this class of architectures.

Acknowledgements

This research was partially sponsored by NSF grants CCF-0540926 and CCF-0702714. The researchers at the Universidad Jaime I of Castellón were supported by the CICYT project TIN2005-09037-C02-02 and FEDER, and projects No. P1-1B2007-19 and P1-1B2007-32 of the Fundación Caixa-Castellón/Bancaixa and UJI. We thank the other members of the FLAME team for their support.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

1. Satish Balay, William Gropp, Lois Curfman McInnes, and Barry Smith. PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, Oct. 1996.
2. S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *Proceedings of PDSEC08*, 2008.
3. S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. FLAG@lab: An M-script API for linear algebra operations on graphics processors. FLAME Working Note #30 Technical Report ICC 01-02-2008, Depto. de Ingenieria y Ciencia de Computadores, Universidad Jaume I, February 2008.
4. 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. Soft.* submitted.
5. Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software*, 31(1):27–59, March 2005.
6. Wesley C. 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.

7. Tom Cwik, Robert van de Geijn, and Jean Patterson. The application of parallel computation to integral equation models of electromagnetic scattering. *Journal of the Optical Society of America A*, 11(4):1538–1545, April 1994.
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. L. Demkowicz, A. Karafiat, and J.T. Oden. Solution of elastic scattering problems in linear acoustics using *h-p* boundary element method. *Comp. Meths. Appl. Mech. Engrg*, 101:251–282, 1992.
10. C. Edwards, P. Geng, A. Patra, and R. van de Geijn. Parallel matrix distributions: have we been doing it all wrong? Technical Report TR-95-40, Department of Computer Sciences, The University of Texas at Austin, 1995.
11. 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.
12. Thierry Joffrain, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Rapid development of high-performance out-of-core solvers. In *Proceedings of PARA04*.
13. Enrique S. Quintana-Ortí and Robert van de Geijn. Updating an lu factorization with pivoting. *ACM Trans. Math. Soft.*
14. 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.
15. Robert A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package*. The MIT Press, 1997.