UNIVERSITAT
JAUME·I

Informe Técnico ICC 01-02-2008

GLAME@lab: An M-script API for Linear Algebra Operations on Graphics Processors

Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí

Febrero de 2008

Departamento de Ingeniería y Ciencia de Computadores

Correo electrónico: {barrachi, castillo, figual, mayo, quintana}@icc.uji.es

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

GLAME@lab: An M-script API for Linear Algebra Operations on Graphics Processors

Sergio Barrachina[1],
Maribel Castillo[2],
Francisco D. Igual[3],
Rafael Mayo[4],
Enrique S. Quintana-Ortí[5],

Abstract:

We propose two high-level application programming interfaces (APIs) to use a graphics processing unit (GPU) as a coprocessor for dense linear algebra operations. Combined with an extension of the FLAME API and an implementation on top of NVIDIA CUBLAS, the result is an efficient and user-friendly tool to design, implement, and execute dense linear algebra operations on the current generation of NVIDIA graphics processors, of wide-appeal to scientists and engineers. As an application of the developed APIs, we implement and evaluate the performance of three different variants of the Cholesky factorization.

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: barrachi@icc.uji.es.

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

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

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

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

GLAME@lab: Interfaz de Programación M-script para Operaciones de Álgebra Lineal Densa sobre Procesadores Gráficos

Sergio Barrachina[6],
Maribel Castillo[7],
Francisco D. Igual[8],
Rafael Mayo[9],
Enrique S. Quintana-Ortí[10],

Resumen:

El artículo presenta dos interfaces de programación (APIs) de alto nivel para el uso de unidades de procesamiento gráfico (GPU) como coprocesadores para la realización de operaciones de álgebra lineal densa. En combinación con una extensión de la interfaz de FLAME y una implementación basada en la biblioteca CUBLAS de NVIDIA, el resultado es una herramienta eficiente y de fácil uso para diseñar, implementar y ejecutar operaciones de álgebra lineal densa sobre la última generación de procesadores gráficos de NVIDIA. Como aplicación de la interfaz desarrollada, se implementa y evalúa el rendimiento de tres variantes de la factorización de Cholesky.

Palabras clave:

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

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

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

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

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

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

GLAME@lab: An M-script API for Linear Algebra Operations on Graphics Processors

Sergio Barrachina   Maribel Castillo   Francisco D. Igual   Rafael Mayo
Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
{barrachi,castillo,figual,mayo,quintana}@icc.uji.es

Abstract

We propose two high-level application programming interfaces (APIs) to use a graphics processing unit (GPU) as a co-processor for dense linear algebra operations. Combined with an extension of the FLAME API and an implementation on top of NVIDIA CUBLAS*, the result is an efficient and user-friendly tool to design, implement, and execute dense linear algebra operations on the current generation of NVIDIA graphics processors, of wide-appeal to scientists and engineers. As an application of the developed APIs, we implement and evaluate the performance of three different variants of the Cholesky factorization.*

Keywords: M-script languages, graphics processors, linear algebra, BLAS, high performance.

1 Introduction

The improvements in performance, functionality, and programmability of the current generation of graphics processors (GPUs) have attracted interest in exploring the use of this class of hardware for general–purpose computation and, particularly, for linear algebra operations [9, 12, 2, 3]. In this line, the development of the CUDA [14] Application Programming Interface (API) is a positive step towards presenting NVIDIA graphics hardware as a general–purpose co–processor. Nevertheless, we believe this class of interfaces still falls a step too short as a vast majority of scientists and engineers employ user–friendly environments like MATLAB, OCTAVE, or LABVIEW to perform complex analysis, modeling, and simulations.

In response to this situation, in this paper we present two high-level APIs to execute the BLAS (*Basic Linear Algebra Subprograms*) functionality [13, 8, 7] on graphics processors from user-friendly environments. Both APIs are based on the popular M-script language that is used in MATLAB/OCTAVE (M-code) and LABVIEW (MATHSCRIPT) but, while the first API leaves the user in control of transferring the data to the GPU memory, the second API transparently takes care of this process at the cost of a certain overhead.

The Formal Linear Algebra Methods Environment (FLAME) encompasses a methodology for deriving provably correct algorithms for dense linear algebra operations as well as an approach to represent (and code) the resulting algorithms [11, 4]. A key observation to FLAME is that in reasoning about algorithms intricate indexing is typically avoided and it is with the introduction of complex indexing that programming errors are often introduced and confidence in code is diminished. Thus, a carefully designed API should avoid explicit indexing whenever possible. FLAME@lab and FLAME/C are examples of such APIs for the M-script and C programming languages [5]. As a second contribution of this paper, we naturally extend FLAME to cover graphics hardware. Using this extension, the design and development of high-performance dense linear algebra codes for this class of hardware results in a significant reduction in effort compared with more traditional approaches to such library development [1, 6].

Our prototype implementation of these APIs on top of NVIDIA CUBLAS offers a measure of the efficacy of this approach (in terms of ease-of-use), and an experimental evaluation on a G80 processor reports on its performance.

The rest of the paper is structured as follows. In Sections 2 and 3 we describe the proposed high-level APIs. In Section 4 we illustrate how these APIs allow easy migration of high-performance linear algebra operations coded using the FLAME@lab API to GPUs. In Section 5 we give some hints on the implementation of the APIs on top of NVIDIA CUBLAS interface. In Section 6 we employ the factorization of a symmetric positive definite matrix to evaluate the performance of the APIs in combination with FLAME. Finally, in Section 7 we give some concluding remarks.

2 An Advanced Interface for Linear Algebra Operations

In this section we present the interface to GLAME@lab, a Graphics Linear Algebra Methods Environment for MATLAB and LABVIEW-like environments. The interface allows the user to initialize and terminate the execution environment, transfer data between main memory and the graphics device memory, and execute the functionality of BLAS on the graphics hardware.

The following attributes describe a matrix as it is stored in the memory of a computer:

1. the datatype of the entries in the matrix,
2. the row and column dimensions of the matrix,
3. the address where the data is stored, and
4. the mapping that describes how the two-dimensional array is mapped to one-dimensional memory.

We note that the memory of a computer includes, among others, main memory and GPU memory space. Hereafter *object* denotes a descriptor of a matrix that is physically stored in the latter space.

2.1 Initializing and finalizing the GLAME@lab API

Before using the environment one must initialize it with a call to

```
GLA_Init( )
```

Purpose: Initialize GLAME@lab.

If no more GLAME@lab calls are to be made, the environment is exited by calling

```
GLA_Finalize( )
```

Purpose: Finalize GLAME@lab.

2.2 Linear algebra objects

The following call creates an object (*descriptor* or *handle*) for a matrix and allocates space to store the entries of the matrix in the GPU memory:

```
A = GLA_Obj_create( datatype, m, n )
```

Purpose: Create an object that describes an $\mathtt{m} \times \mathtt{n}$ matrix `A`, with entries of type `datatype`, and create the associated storage array in the GPU memory space.

Here `datatype` is a string that can take on the values `'GLA_FLOAT'` and `'GLA_COMPLEX'` for the obvious (single-precision) real and complex datatypes that are commonly encountered. (No support is provided yet for double-precision numbers as current hardware in general does not operate with this type of data.) The leading dimension of the array that is used to store the matrix is determined inside of this call; GLAME@lab treats vectors as special cases of matrices: an $n \times 1$ matrix or a $1 \times n$ matrix.

If an object is created with `GLA_Obj_create`, a call to `GLA_Obj_free` is required to ensure that all space associated with the object in the GPU memory is properly released:

```
GLA_Obj_free( A )
```

Purpose: Free all space allocated to store data associated with `A` in the GPU memory.

2.3 Inquiry routines

A number of inquiry routines can be used to access information about an object. The datatype and row and column dimensions of the matrix can be extracted by calling

```
datatype = GLA_Obj_datatype( A )
m        = GLA_Obj_length  ( A )
n        = GLA_Obj_width   ( A )
```

Purpose: Extract datatype, row, or column dimension of matrix `A`, respectively.

2.4 Matrix contents

The contents of an object can be initialized to a certain scalar value with the call:

```
GLA_Obj_set( A, value )
```

Purpose: Initialize all entries of matrix `A` in the GPU memory space to equal `value`.

Filling the contents of an object (transferring data from main memory to GPU memory space) can be done using

```
GLA_Axpy_matrix_to_object( alpha, B, A, i, j )
```

Purpose: Fill the entries of the matrix `A` in the GPU memory space with the result of the product `alpha`·`B`.

Given a matrix `B` with `m` rows and `n` columns and a scalar `alpha`, both in main memory, the previous call is equivalent to

```
A( i:i+m-1, j:j+n-1 ) = alpha * B + A( i:i+m-1, j:j+n-1 );
```

where A is an object in GPU memory space.

The call with the opposite purpose (transferring data from GPU memory space to main memory) is

```
B = GLA_object_to_matrix( m, n, A, i, j )
```

Purpose: Retrieve the entries of the matrix `A` in the GPU memory space.

Upon execution of this call, matrix `B` in main memory is set as

```
B = A( i:i+m-1, j:j+n-1 );
```

2.5 A most useful utility routine

Likely one of the more useful tools in the GLAME@lab library, which is particularly helpful for testing, is

```
GLA_Obj_show( A );
```

Purpose: Print the contents of `A`.

In particular, the result of

```
GLA_Obj_show( A );
```

produces the usual MATLAB-like output:

```
A = [
    < entries_of_A >
];
```

2.6 Views

In GLAME@lab we deal with blocks of a matrix that resides in the GPU memory space by introducing the notion of a *view*, which is a *reference* into an existing matrix or vector. Given an object `A` describing a matrix in the GPU memory space, the following call creates a view of the object:

```
Aview = GLA_Obj_view( A, m, n, i, j )
```

Purpose: Create the view `Aview` consisting of the `m`×`n` submatrix (block) of `A` starting at coordinate (`i`,`j`).

Thus, after the call

```
        Aview = GLA_Obj_view( A, m, n, i, j );
```

`Aview` and `A( i:i+m-1, j:j+n-1 )` refer to the same positions in the GPU memory space. Subsequent modifications of the contents of the view affect the original contents of the matrix.

2.7 Computational kernels

There exists a GLAME@lab routine for each subprogram defined in the BLAS interface, organized following the usual Level 1, 2, and 3 structure. Below we give a short specification of these routines.

2.7.1 Level 1 BLAS

In the following list of calls, `x` and `y` are vectors in the GPU memory space while `alpha` is a scalar. The functionality of the routine is obvious from the relation between the name of the Level 1 GLAME@lab routine and the BLAS.

```
        GLA_Swap  (        x, y )
        GLA_Scal  ( alpha, x )
        GLA_Copy  (        x, y )
        GLA_Axpy  ( alpha, x, y )
alpha = GLA_dot   (        x, y )
alpha = GLA_nrm2  (        x )
alpha = GLA_asum  (        x )
iota  = GLA_iamax(         x )
```

With this interface the dimensions of the objects the routine operates on are not specified. If only a part of a vector is involved in an operation, this can be accomplished using a view of the object. Also, there is no need to use different calls depending on the datatypes of the entries as this information is embedded in the object.

2.7.2 Level 2 BLAS

In the following Level 2 calls, `x` and `y` are vectors and `A` is a matrix, all in the GPU memory space; `alpha` and `beta` are scalars.

```
GLA_Gemv(       trans,       alpha, A, x, beta, y )
GLA_Symv( uplo,              alpha, A, x, beta, y )
GLA_Trmv( uplo, trans, diag,        A, x )
GLA_Trsv( uplo, trans, diag,        A, x )

GLA_Ger (                    alpha, x, y, A )
GLA_Syr ( uplo,              alpha, x,    A )
GLA_Syr2( uplo,              alpha, x, y, A )
```

There are three mode parameters (options), in the Level 2 calls which can take on values from the following lists of strings.

1. `trans`: `'GLA_NO_TRANSPOSE'`, `'GLA_TRANSPOSE`’ (also `'GLA_CONJ_TRANSPOSE'` for complex matrices);
2. `uplo`: `'GLA_UPPER_TRIANGULAR'`, `'GLA_LOWER_TRIANGULAR`’;
3. `diag`: `'GLA_NONUNIT_DIAG'`, `'GLA_UNIG_DIAG`’.

These parameters will reappear next, in the Level 3 calls, where they can take on the same values.

2.7.3 Level 3 BLAS

The list of Level 3 calls involves matrices in the GPU memory space as `A`, `B`, `C`; and scalars like `alpha` and `beta`.

```
GLA_Gemm(             transA, transB,       alpha, A, B, beta, C )
GLA_Symm(             transA, transB,       alpha, A, B, beta, C )
GLA_Syrk(       uplo, trans,                alpha, A,    beta, C )
GLA_Trmm( side, uplo, transA,         diag, alpha, A,          B )
GLA_Trsm( side, uplo, transA,         diag, alpha, A,          B )
```

A new mode parameter appears here, `side`, which can take on the strings `'GLA_LEFT'`, `'GLA_RIGHT'`.

2.8 Putting it all together: a basic code to multiply two matrices

Figure 1 gives a fragment of M-script code that uses the routines in the GLAME@lab interface to compute the product $C := A^T \cdot B$, where A, B and C are matrices with real entries of dimension $k \times m$, $k \times n$ and $m \times n$, respectively.

Figure 1. GLAME@lab code to compute the product $C := A^T B$.

3 A Simple Interface for Linear Algebra Operations

For users who do not want to deal with the management and transference of objects between main memory and GPU memory space, we propose a simplified interface, built on top of GLAME@lab, which hides this process.

This interface only provides access to the BLAS kernels, receiving as input parameters matrices that are stored in the main memory. Each time one of the routines in this interface is invoked, the entries of the input matrices are transfered to the GPU memory space, operation proceeds there, and the results are returned to the main memory in the form of output parameters. We note that this interface, while being easier to use, introduces a considerable overhead when the purpose is to use the GPU to perform multiple operations on a matrix, as the data will need to be transfered once for each operation that is performed on it.

The routine names of this simplified interface only differ from those of the previous API in the prefix, which corresponds now to "`GLAS_`"; thus, e.g., the routines that compute the matrix multiplication $C := \beta \cdot C + \alpha \cdot A \cdot B$ using both interfaces are:

```
GLA_Gemm( transA, transB,    GLAS_Gemm( transA, transB,
          alpha, A, B,                  alpha, A, B,
          beta,  C )                    beta,  C )
```

While the appearance is similar we note that in the call to `GLA_Gemm`, `A`, `B` and `C` are objects for matrices in the GPU memory space, while in the call to `GLAS_Gemm` these refer to matrices in the main memory.

The code that calculates the product $C := A^T \cdot B$, with A, B and C real matrices of dimension $k \times m$, $k \times n$ and $m \times n$, respectively, is given in Figure 2.

4 Porting FLAME to Graphics Processors

FLAME avoids complicate indexing by embedding the notion of a *view*, which is a reference into an existing matrix or vector, into a *partitioning* operation. Figure 3 illustrates the use of the FLAME@lab API to code a blocked algorithm that computes the Cholesky factorization of a (symmetric positive definite) matrix A [10]. In this operation, the matrix is decomposed into the product $A = LL^T$, where the lower triangular matrix L is known as the Cholesky factor of A. Upon completion of the code, the entries of the Cholesky factor overwrites the corresponding entries of the lower triangular part of A.

```
A = read_matrix( k, m );  % 'Read' input matrices A,B
B = read_matrix( k, n );  % user-supplied read_matrix
C = zeros( m, n );

GLAS_Init();              % Initialize environment

GLAS_Gemm( 'GLA_TRANSPOSE',...
           'GLA_NO_TRANSPOSE',...
           1.0, A, B, 0.0, C );               % Compute C:=A^T B

disp( C );                % Print out results

GLAS_Finalize();          % Free environment
```

Figure 2. GLASME@lab code to compute the product $C := A^T B$.

In this section we follow this successful approach extending FLAME to graphics processors. The partitioning and repartitionings in the code are just indexing operations that do not modify the contents of the matrix. The specific behaviour of these operations is explained in detail in [5], but will also become evident from the presentation of the GLAME@lab interface that is given next.

Given a descriptor `A` of a matrix in the GPU memory space, the following call creates descriptors (or views) of the four quadrants:

```
[ ATL, ATR,...
  ABL, ABR     ] = GLA_Part_2x2( A, mb, nb, quadrant )
```

Purpose: Partition matrix `A` into four quadrants where the quadrant indicated by `quadrant` is `mb` × `nb`.

Here `quadrant` is a string that can take on the values `'GLA_TL'`, `'GLA_TR'`, `'GLA_BL'`, and `'GLA_BR'` to indicate that `mb` and `nb` specify the dimensions of the Top-Left, Top-Right, Bottom-Left, or Bottom-Right quadrant, respectively.

Thus, invocation of the operation

```
[ATL, ATR,...
 ABL, ABR     ] = GLA_Part_2x2( A, mb, nb, 'GLA_TL' );
```

in GLAME@lab creates four views, one for each quadrant. Subsequent modifications of the contents of a view affect the original contents of the matrix in the GPU memory space.

A 2 × 2 partitioning can be further divided into a 3 × 3 partitioning using the call

```
[ A00, A01, A02,...
  A10, A11, A12,...
  A20, A21, A22     ] =  GLA_Repart_2x2_to_3x3( ATL, ATR,...
                                                ABL, ABR,...
                                                mb, nb,...
                                                quadrant )
```

Purpose: Repartition a 2 × 2 partitioning of matrix `A` into a 3 × 3 partitioning where the `mb` × `nb` submatrix `A11` is split from the quadrant indicated by `quadrant`.

Here `quadrant` can again take on the values `'FLA_TL'`, `'FLA_TR'`, `'FLA_BL'`, and `'FLA_BR'` to indicate that the `mb` × `nb` submatrix `A11` is split from submatrix `ATL`, `ATR`, `ABL`, or `ABR`, respectively.

Once the contents of the so-identified submatrices have been updated, the blocks of the 3 × 3 partitioning can be merged back into a 2 × 2 partitioning by a call to

```
[ ATL, ATR,...
  ABL, ABR     ] = FLA_Cont_with_3x3_to_2x2( A00, A01, A02,...
                                             A10, A11, A12,...
                                             A20, A21, A22,...
                                             quadrant )
```

Purpose: Update the 2 × 2 partitioning of matrix `A` by moving the boundaries so that `A11` is joined to the quadrant indicated by `quadrant`.

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

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

  while ( size( ATL, 1 ) < size( A, 1 ) )

    b = min( size( ABR, 1 ), nb_alg );

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

    %------------------------------------------------------------%
    A11 = FLA_Cholesky_unb( A11 );
    A21 = A21 * inv( tril( A11 ) )';
    A22 = A22 - tril( A21 * A21' );
    %------------------------------------------------------------%

    [ 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, ATR
            ABL, ABR ];

return
```

Figure 3. FLAME@lab blocked code to compute the Cholesky factorization.

This time the value of quadrant ('FLA_TL', 'FLA_TR', 'FLA_BL', or 'FLA_BR') indicates to which quadrant the submatrix A11 is to be joined.

Using these routines, we can easily implement a code to compute the Cholesky factorization using the GLAME@lab, as illustrated in Figure 4.

We note two subtle differences between the FLAME@lab and GLAME@lab codes for the Cholesky factorization (Figures 3 and 4). First, the GLAME@lab code operates on a matrix which already resides on the GPU memory space; this implies that the environment has been initialized, space for this matrix has been allocated in the GPU memory, and the data has been already transferred to this space (see, e.g., Figure 1). For the FLAME@lab code, the only requirement before calling the routine is the initialization of the matrix entries with the appropriate values. Second, updates on submatrices like ATL in the FLAME@lab code do not modify the contents of the original matrix A. This is a fundamental difference with GLAME@lab where ATL is a view into A and, therefore, any modification of the two objects affects the other (sub)matrix.

Similar routines exist that provide 2×1 and 1×2 partitionings, repartition these into 3×1 and 1×3 partitionings and merge them back. The following two subsections provides a summarized list of these routines. For more details, see [5].

4.1 Horizontal and vertical partitionings

In addition to the routines listed above, GLAME@lab completes its FLAME compatibility with a set of horizontal partitioning routines:

```
function [ A ] = GLA_Cholesky_blk( A, nb_alg )

  [ ATL, ATR, ...
    ABL, ABR ] = GLA_Part_2x2( A, 0, 0, ’GLA_TL’ );

  while ( GLA_Obj_length( ATL, 1 ) < GLA_Obj_length( A, 1 ) )

    b = min( GLA_Obj_length( ABR, 1 ), nb_alg );

    [ A00, A01, A02, ...
      A10, A11, A12, ...
      A20, A21, A22 ] = GLA_Repart_2x2_to_3x3( ATL, ATR, ...
                                               ABL, ABR, ...
                                               b, b, ’GLA_BR’ );

    %------------------------------------------------------------%
    GLA_Cholesky_unb( A11 );
    % A21 = A21 * inv( tril( A11 ) )’;
    GLA_Trsm( ’GLA_RIGHT’, ’GLA_LOWER_TRIANGULAR’,
              ’GLA_TRANSPOSE’, ’GLA_NONUNIT_DIAG’,
              1.0, A11, A21 );
    % A22 = A22 - tril( A21 * A21’ );
    GLA_Syrk( ’GLA_LOWER_TRIANGULAR’, ’GLA_NO_TRANSPOSE’,
              -1.0, A21, 1.0, A22 );
    %------------------------------------------------------------%

    [ ATL, ATR, ...
      ABL, ABR ] = GLA_Cont_with_3x3_to_2x2( A00, A01, A02, ...
                                             A10, A11, A12, ...
                                             A20, A21, A22, ...
                                             ’GLA_TL’ );

  end

return
```

Figure 4. GLAME@lab blocked code to compute the Cholesky factorization.

```
[ AT,...
  AB     ] = FLA_Part_2x1            ( A,...
                                       mb, side )
[ A0,...
  A1,...
  A2     ] = FLA_Repart_2x1_to_3x1   ( AT,...
                                       AB,...
                                       mb, side )
[ AT,...
  AB     ] = FLA_Cont_with_3x1_to_2x1( A0,...
                                       A1,...
                                       A2,...
                                           side )
```

Here, `side` can take on the values `’GLA_TOP’`,`’GLA_BOTTOM’`.

Similar routines are implemented for vertical partitioning schemes:

```
[ AL, AR ]     = FLA_Part_1x2            ( A,...
                                           nb, side )
[ A0, A1, A2 ] = FLA_Repart_1x2_to_1x3   ( AL, AR,...
                                           nb, side )
[ AL, AR ]     = FLA_Cont_with_1x3_to_1x2( A0, A1, A2,...
                                               side )
```

Here, `side` can take on the values `'GLA_LEFT'`, `'GLA_RIGHT'`.

5 Implementation of GLAME@lab on top of CUBLAS

We have developed a prototype implementation of the previous interface on top of NVIDIA CUBLAS. This API provides wrappers to help writing Fortran programs that use the library. To illustrate the Fortran API, Figure 5 shows a fragment of Fortran code that uses this interface to compute $C := A^T \cdot B$, with A, B and C real matrices of dimension $k \times m$, $k \times n$ and $m \times n$, respectively.

```
#define SIZEOF_REAL 4

*     'Read' input matrices A,B; user-supplied read_matrix
      CALL READ_MATRIX( K, M, A, LDA )
      CALL READ_MATRIX( K, N, B, LDB )

*     Initialize environent
      CALL CUBLAS_INIT

*     Create space for objects A,B,C in the GPU
      STAT = CUBLAS_ALLOC( K*M, SIZEOF_REAL, AOBJ )
      STAT = CUBLAS_ALLOC( K*N, SIZEOF_REAL, BOBJ )
      STAT = CUBLAS_ALLOC( M*N, SIZEOF_REAL, COBJ )

*     Set contents of objects A,B,C in the GPU
      CALL CUBLAS_SET_MATRIX( K, M, SIZEOF_REAL, A, LDA, AOBJ, K )
      CALL CUBLAS_SET_MATRIX( K, N, SIZEOF_REAL, B, LDB, BOBJ, K )

*     Compute C:=A^T B
      CALL CUBLAS_GEMM( 'Transpose', 'No Transpose',
                        M, N, K, 1.0, AOBJ, K, BOBJ, K,
                                 0.0,  COBJ, M )

*     Print out results, user-supplied print_matrix
      CALL CUBLAS_GET_MATRIX( M, N, SIZEOF_REAL, COBJ, M, C, LDC )
      CALL PRINT_MATRIX( M, N, C, LDC )

*     Free objects A,B,C in the GPU
      CALL CUBLAS_FREE( AOBJ )
      CALL CUBLAS_FREE( BOBJ )
      CALL CUBLAS_FREE( COBJ )

*     Free environment
      CALL CUBLAS_SUTDOWN
```

Figure 5. CUBLAS **code to compute the product** $C := A^T B$**.**

While there are strong similarities between our advanced GLAME@lab API and the CUBLAS API (compare Figures 1 and 5), we believe our interface to be much more intuitive when developing codes for complex linear algebra operations like, e.g., the Cholesky factorization (compare Figures 4 and 6).

6 Exploring the Performance of the APIs

We next evaluate the performance of several alternatives to compute the Cholesky factorization of a symmetric definite positive matrix using a CPU and a graphics processor. Our purpose is to combine the ease of use of FLAME and the high performance characteristic of the modern graphics processors. Three different implementations of the Cholesky factorization have been implemented ([10]). Unblocked and blocked algorithms for the different variants are given in Figure 7 in a notation that has been developed as part of the FLAME project [11, 4].

Each variant has been implemented in three ways:

- Using NVIDIA CUBLAS library from a Fortran program, as in Figure 6.

```
#define SIZEOF_REAL 4
#define IDX2F(I,J,LDA) ((((J)-1)*(LDA))+((I)-1))*SIZEOF_REAL
#define AOBJ(I,J) DEVPTRA+IDX2F(I,J,LDA)

*
*     Compute the Cholesky factorization A = L*L'.
*
      DO 20 J = 1, N, NB
*
*        Update and factorize the current diagonal block and test
*        for non-positive-definiteness.
*
         JB = MIN( NB, N-J+1 )
         CALL SPOTF2( 'Lower', JB,
     $                AOBJ(J,J), LDA, INFO )
         IF( INFO.NE.0 )
     $      GO TO 30
         IF( J+JB.LE.N ) THEN
*
*           Compute the current block column.
*
            CALL CUBLAS_STRSM( 'Right', 'Lower',
     $                         'Transpose', 'Non-unit',
     $                         N-J-JB+1, JB,
     $                         ONE, AOBJ(J,J),      LDA,
     $                               AOBJ(J+JB, J), LDA )
            CALL CUBLAS_SSYRK( 'Lower', 'No transpose',
     $                         N-J-JB+1, JB,
     $                         -ONE, AOBJ(J+JB,J),     LDA,
     $                           ONE, AOBJ(J+JB,J+JB), LDA )
         END IF
   20 CONTINUE

```

Figure 6. CUBLAS blocked code to compute the Cholesky factorization.

- Using the advanced interface GLAME@lab from an OCTAVE M-script program, as in Figure 4.
- Using the simplified interface GLAME@lab from an OCTAVE M-script program.

In addition, we propose a hybrid approach, in which CPU and GPU work together to compute the result, and also evaluate the performance of the Cholesky routine implementation in MATLAB/OCTAVE.

The implementations have been tested on an Intel Core2 Duo processor (codename Crusoe E6320) on the CPU side, and a Nvidia Geforce 8800 Ultra (G80 processor) on the GPU side. We have developed Fortran 77 implementations of the unblocked and blocked algorithms linked with CUDA 1.1 (with the same version of CUBLAS library) for the GPU. In the CPU, we employed LAPACK version 3.0 and GotoBLAS version 1.19. The compilers include GNU Fortran Compiler version 3.3.5 and NVCC (NVIDIA compiler) release 1.0, version 0.2.1221. All codes have been tested on OCTAVE version 2.9.19.

All the results on the GPU presented hereafter include the time required to transfer the data from the main memory to the GPU memory and retrieve the results back. Results are reported in terms of GFLOPS (10^9 flops per second). A single core of the Intel processor was employed in the experiments.

Figure 8 (left-side) presents a comparison between the three implemented variants of the Cholesky factorization using the advanced GLAME@lab interface, including also the performance obtained for the MATLAB/OCTAVE implementation of the factorization routine (function `chol`). All three GLAME@lab implementations deliver higher performance than the OCTAVE CPU based implementation. It is also important to note how the performance of the GPU implementations is higher when matrix sizes are large; on the other side, the CPU implementation keeps the performance constant for all matrix sizes, attaining higher performance than the GPU for small matrices.

When using the simple implementation (GLAS), transfer times between main and video memory play a determinant role on the final performance results. Figure 8 (right-side) show the performance of the three implemented variants for the Cholesky factorization, comparing them with the FLAME@lab implementations of the same routines. The results show that

```
Algorithm: $A := \textsc{Chol\_unb}(A)$
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
    $\left(\begin{array}{c|c} A_{TL} & A_{TR} \\ \hline A_{BL} & A_{BR} \end{array}\right) \rightarrow \left(\begin{array}{c|c|c} A_{00} & a_{01} & A_{02} \\ \hline a_{10}^{\mathrm{T}} & \alpha_{11} & a_{12}^{\mathrm{T}} \\ \hline A_{20} & a_{21} & A_{22} \end{array}\right)$
        where $\alpha_{11}$ is $1 \times 1$

    Variant 1:
        $\alpha_{11} := \sqrt{\alpha_{11}}$
        $a_{21} := a_{21}/\alpha_{11}$
        $A_{22} := A_{22} - a_{21}a_{21}^{\mathrm{T}}$
    Variant 2:
        $a_{10}^{\mathrm{T}} := a_{10}^{\mathrm{T}}\mathrm{TRIL}\left(A_{00}\right)^{-\mathrm{T}}$
        $\alpha_{11} := \alpha_{11} - a_{10}^{\mathrm{T}}a_{10}$
        $\alpha_{11} := \sqrt{\alpha_{11}}$
    Variant 3:
        $\alpha_{11} := \alpha_{11} - a_{10}^{\mathrm{T}}a_{10}$
        $\alpha_{11} := \sqrt{\alpha_{11}}$
        $a_{21} := a_{21} - A_{20}a_{10}$
        $a_{21} := a_{21}/\alpha_{11}$

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

```
Algorithm: $A := \textsc{Chol\_blk}(A)$
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 $n_b \times n_b$

    Variant 1:
        $A_{11} := \textsc{Chol\_unb}(A_{11})$
        $A_{21} := A_{21}\mathrm{TRIL}\left(A_{11}\right)^{-\mathrm{T}}$
        $A_{22} := A_{22} - A_{21}A_{21}^{\mathrm{T}}$
    Variant 2:
        $A_{10} := A_{10}\mathrm{TRIL}\left(A_{00}\right)^{-\mathrm{T}}$
        $A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}}$
        $A_{11} := \textsc{Chol\_unb}(A_{11})$
    Variant 3:
        $A_{11} := A_{11} - A_{10}A_{10}^{\mathrm{T}}$
        $A_{11} := \textsc{Chol\_unb}(A_{11})$
        $A_{21} := A_{21} - A_{20}A_{10}^{\mathrm{T}}$
        $A_{21} := A_{21}\mathrm{TRIL}\left(A_{11}\right)^{-\mathrm{T}}$

    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
```

Figure 7. Multiple variants of the unblocked (left) and blocked (right) algorithms for the Cholesky factorization implemented on top of GLAME@lab.

GPU implementations obtain better performance than CPU/FLAME based implmementations, and even obtain similar results than the MATLAB/OCTAVE implmementations shown in Figure 8 (left-side).

GPU implementations work better when they process big amounts of data. For small matrices, Figure 8 (left-side) shows a higher performance when the factorization process is performed on the CPU. Therefore, we introduce a hybrid algorithm that computes the small factorizations of the diagonal blocks on the CPU, avoiding the overhead introduced by extra transfers and operations that do not match easily the GPU architecture (basically, the square root calculation). Figure 9 shows the performance of this hybrid approach for the first variant of the Cholesky implementation.

As the results attained for the hybrid implementation are better than those obtained for the "pure" GPU implementations, we use them to compare the GLAME@lab performance with the CUBLAS Fortran implementations. The low performance obtained by GLAME@lab implementations compared with that of the CUBLAS Fortran implementations, see Figure 9, can be explained in two ways:

1. The interpreted nature of the M-script is presented as one of the main bottlenecks when trying to achieve high performance on these type of implementations. Although there are no memory copies when the GLAME@lab repartition routines are invoked, the high amount of calls that have to be interpreted is a key factor that limits the final performance of the implementations. As shown in Figure 10 (left-side), the repartition overhead on the overall execution time is considerable for these type of blocked implementation.

2. In addition to transfer times, there exists another source of inefficiency in GLAME@lab BLAS implementations. Figure 10 (right-side) shows the difference in performance between CUBLAS `TRSM` implementation and the same routine executed through GLAME@lab. Two facts limit the final performance:

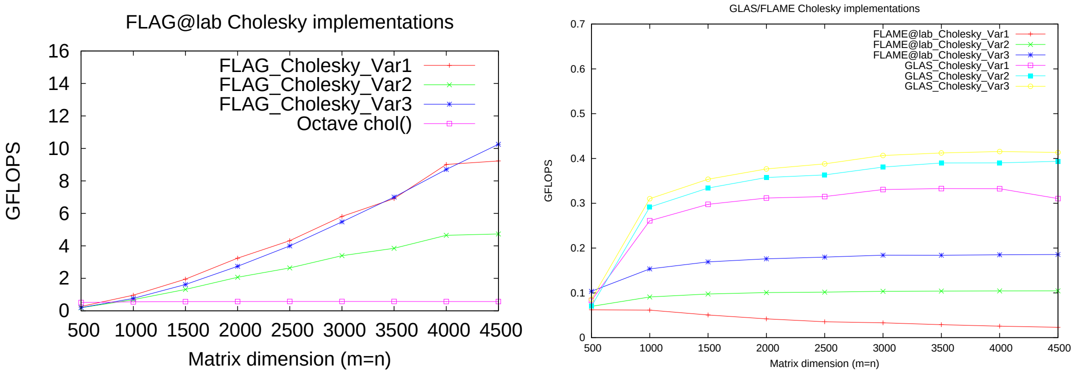

Figure 8. Comparison between the Cholesky factorization implementations in OCTAVE and three different implementations of the same routine accelerated through GLAME@lab (left-side). Comparison between the performance of the simplified GLAME@lab interface (GLAS) and the same implementations using only the CPU through the FLAME@lab API (right-side).

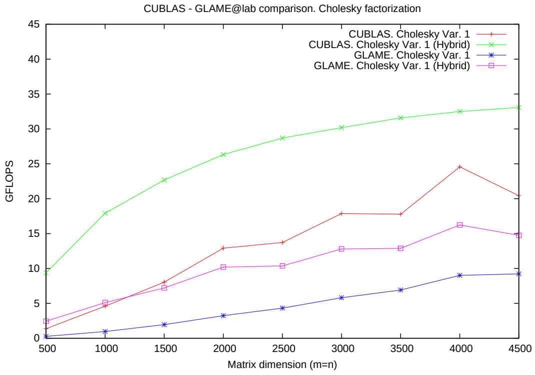

Figure 9. Comparison between the performance attained for a CUBLAS implementation of Variant 1 of the Cholesky factorization, and the same implementation based on GLAME@lab. Both are executed exclusively on GPU and symultaneously on CPU and GPU (hybrid approach).

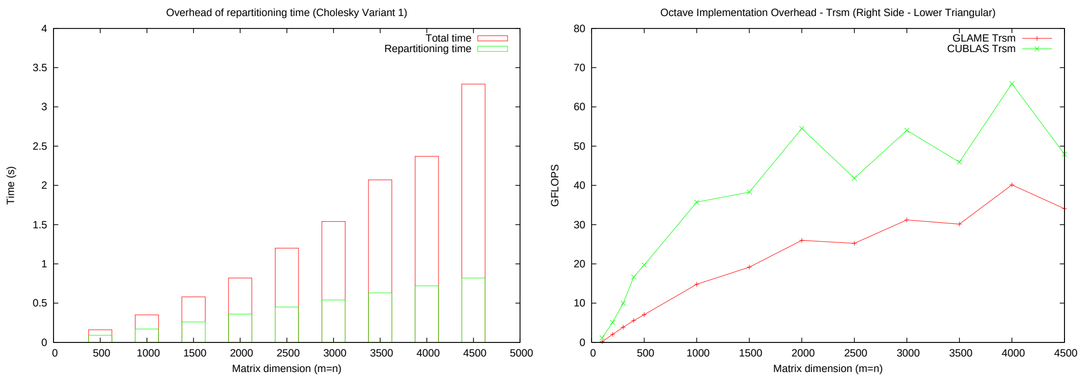

Figure 10. On the left side, overhead introduced by the repartitioning procedures characteristic of FLAME@lab/GLAME@lab. On the right side, difference in performance between a basic BLAS execution based on Fortran CUBLAS and GLAME@lab implementation based on MEX-files.

- The internal implementation of the MEX-files interface of the tested version of Octave introduces a considerable overhead when invoking and returning from a MEX routine.
- The internal double-precision data representation of Octave must be converted into single-precision before operating with it on GPU, and transformed back into double-precision before the final transfer to main memory. This implies an important penalty on the overall performance of this type of routines.

A more detailed analysis of the performance of Fortran implementations for the Cholesky factorization on top of CUBLAS is given in [3].

7 Concluding Remarks

This paper makes the following contributions and observations:

- We have proposed two API to assist in the development of M-script codes for complex dense linear algebra operations, targeting performance and ease-of-use. The advanced APIs in GLAME@lab requires more intervention from the user but pays off in terms of performance when the API is used to implement a code that performs multiple operations on a matrix (or parts of it, as in blocked algorithms).
- We have also described an extension of FLAME that enables a straight-forward translation of codes written using FLAME@lab to graphics processors. Thus, our approach inherits the advantages of FLAME, which include a cleaner notation, a formal derivation procedure of provably correct algorithms, and the existence of a library of codes for more complex dense linear algebra operations than those provided by BLAS.
- We have developed a prototype implementation of the proposed APIs on top of NVIDIA CUBLAS to show the validity of this approach.
- Our experimental evaluation on the NVIDIA G80 graphics processor shows that the advanced interface is an interesting approach to achieve high performance on MATLAB/OCTAVE code for linear algebra operations while simultaneously taking benefit from the advantadges of FLAME.
- Although the reported performance is not comparable with that of CUBLAS native implementations, GLAME@lab is presented as a competitive approach that combines a user-friendly environment with a high performance computing platform.

Our proposal is a first step towards the introduction of graphics processors into FLAME in order to achieve high performance with low cost hardware and an easy environment. Future work will include the adaptation of the FLAME/C API to graphics processors, trying to achieve all the performance that a modern GPU can offer, without the penalties of the interpreted languages.

Acknowledgments

This research was partially supported by the CICYT project TIN2005-09037-C02-02 and FEDER, and project No. P1-1B2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI. Francisco D. Igual is supported as well by a research fellowship from the *Universidad Jaume I of Castellón* (PREDOC/2006/02).

References

[1] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' guide (third ed.)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1999.

[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. To appear in proceedings of PDSEC08, 2008.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. Technical report, Universitat Jaume I, 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.*, 31(1):1–26, March 2005.

[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 Trans. Math. Soft.*, 31(1):27–59, March 2005.

[6] 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.

[7] 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.

[8] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Soft.*, 14(1):1–17, March 1988.

[9] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[10] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, Baltimore, 2nd edition, 1989.

[11] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Soft.*, 27(4):422–455, December 2001.

[12] Jin Hyuk Junk and Dianne P. O'Leary. Cholesky decomposition and linear programming on a GPU. Master's thesis, University of Maryland, College Park.

[13] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Trans. Math. Soft.*, 5(3):308–323, Sept. 1979.

[14] NVIDIA. *Nvidia CUDA Compute Unified Device Architecture. Programming Guide*. 2007.