Fast Development of Dense Linear Algebra Codes on Graphics Processors

M. Jesús Zafont, Alberto Martín, Francisco Igual, and Enrique S. Quintana-Ortí
Depto. de Ingeniería y Ciencia de los Computadores
Universidad Jaume I, Castellón (Spain)
E-mails: al051631@uji.es, {martina, figual, quintana}@icc.uji.es

***Abstract*—We present an application programming interface (API) for the C programming language that facilitates the development of dense linear algebra algorithms on graphics processors applying the FLAME methodology. The interface, built on top of the NVIDIA CUBLAS library, implements all the computational functionality of the FLAME/C interface. In addition, the API includes data transference routines to explicitly handle communication between the CPU and GPU memory spaces. The flexibility and simplicity-of-use of this tool are illustrated using a complex operation of dense linear algebra: the Cholesky factorization. For this operation, we implement and evaluate all existing variants on an NVIDIA G80 processor.**

***Index Terms*—Graphics processors, FLAME, linear algebra, high performance.**

I. Introduction

With the advent of remarkably faster graphics processing units (GPUs), these platforms have emerged as a low cost alternative for many scientific and industrial applications. The introduction of simplified interfaces for the development of general purpose algorithms, such as NVIDIA CUDA [1] or AMD Brook+ [2], have consolidated the graphics architecture as an interesting platform for the execution of algorithms with high performance demands. However, it is still difficult to develop reliable algorithms for the GPU combining both ease-of-use, reliability, and high performance.

The present article pursues two major goals: first, the introduction of a high-level API that extends the FLAME project [11], enabling fast development of high-performance reliable codes for computing dense linear algebra operations on the GPU. Second, the evaluation of the performance of the interface using an important operation in the solution of linear systems: the Cholesky factorization. In this operation, a symmetric positive definite (SPD) matrix A is decomposed into the product $A = LL^T$, where L is a lower triangular matrix with the same dimensions as those of A [13]. The choice of this routine can be seen as a vehicle to identify techniques and extract conclusions that can be easily applied to other dense linear algebra operations.

The rest of the article is structured as follows. Section II reviews the key routines of the FLAG/C interface, illustrating their use by means of a simple example. Section III describes the implementation of several variants for the Cholesky factorization using the proposed interface. The codes corresponding to these variants are evaluated next in Section IV. Finally some concluding remarks are summarized in Section V.

II. The FLAG/C API

The primary goal of the FLAG/C interface is to allow library developers to extract much of the performance of current graphics processors in the solution of dense linear algebra problems. In order to do so, FLAG/C abstracts the programmer from the underlying (graphical) architecture. Furthermore, the interface offers the tools to tune the programs while maintaining simplicity of programming. The set of routines provided by the interface is thus a trade-off between simplicity-of-use and flexibility.

FLAG/C, as the rest of the APIs from the FLAME project, follows an object-oriented programming style similar to that of MPI [5], PETSc [6], and PLAPACK [10]. This paradigm makes it possible to hide programming details which usually entail most of the errors through the development process (basically the intricate indexing present in most traditional linear algebra codes and aspects related to the storage format of the matrices).

FLAG/C employs objects to represent matrices and vectors. An object in FLAG/C is defined with the FLAG_Obj datatype, and is implemented internally as a structure with the following attributes:

- data type of the elements of the object,
- object dimensions (number of rows and columns),
- address of element (1,1) of the object in the GPU memory, and
- data layout of the bi-dimensional array for the object elements in the GPU memory.

To illustrate the functionality and usage of the FLAG/C API, we first offer a simple code example which prepares the environment for the Cholesky factorization of a SPD matrix. Figure 1 shows a sequence of steps which perform the following operations:

- line 10: before any FLAG/C invocation the environment is initialized with routine FLAG_Init.
- line 19: the object A that will hold the matrix is created in the GPU memory.
- line 22: data stored in a local buffer in the CPU memory is transferred to the GPU memory using routine FLAG_set_from_buffer.
- line 28: calculation of the Cholesky factorization with the corresponding routine. The code of this routine will be shown later.

- `line 34`: (optionally) transfer result back to CPU memory with routine `FLAG_get_from_Obj`.
- `line 37`: memory space allocated for the object in GPU memory is released with routine `FLAG_Obj_free`.
- `line 40`: Close the environment with routine `FLAG_Finalize`.

```
1  int main(){
2
3    /* GPU object */
4    FLA_Obj A;
5
6    /* Host buffer */
7    float * h_A;
8
9    /* Initialize environment */
10   FLA_Init();
11
12   printf( "Enter matrix dimension:" );
13   scanf( "%d", &n );
14
15   /* Create and fill buffers on CPU */
16   /* ... */
17
18   /* Create and fill buffers on GPU */
19   FLAG_Obj_create( FLAG_FLOAT, n, n, &A );
20
21   /* Perform the GPU->GPU transfers */
22   FLAG_Obj_set_from_buffer( (void*)h_A, &A );
23
24   /* Show the contents of the matrix */
25   FLAG_Obj_show( "A = [", A, "%lf", "]" );
26
27   /* Invoke the Cholesky factorization */
28   FLAG_Chol_blk( A );
29
30   /* Do some other operations */
31   FLAG_Obj_show( "A = [", A, "%lf", "]" );
32
33   /* Transfer back to CPU */
34   FLAG_get_from_Obj( &A, (void*)h_A );
35
36   /* Free buffers */
37   FLAG_Obj_free( &A );
38
39   /* Finalize environment */
40   FLA_Finalize();
41
42 }
```

Fig. 1. Generic `FLAG/C` driver for the invocation of the Cholesky factorization routine

Figure 2 implements a blocked variant of the Cholesky factorization using the `FLAG/C` API. The equivalent algorithm expressed using the `FLAME` notation is shown in Figure 3 (see variant 3 on the right side of the Figure). The notation $m(B)$ in the algorithm refers to the row dimension of B. We believe the rest of the notation is intuitive; for further details, see [14], [15].

`FLAG/C` has a complete set of routines to partition and repartition objects. A comparison of Figures 2 and 3 (right) illustrates how the routines in `FLAG/C` for the partitioning (`FLAG_Part_2x2`), repartitioning (`FLAG_Repart_2x2_to_3x3`), and redistribution of the partitions (`FLAG_Cont_with_3x3_to_2x2`) track the movement of the thick lines in the algorithm.

`FLAG/C` also provides a full interface for the three levels of the *Basic Linear Algebra Subprograms* (BLAS); see, e.g., lines 22–26 of Figure 2. These routines are implemented as

```
1  int FLAG_Chol_blk( FLA_Obj A, int nb_alg )
2  {
3    FLAG_Obj     ATL, ATR,   A00, A01, A02,
4                 ABL, ABR,   A10, A11, A12,
5                             A20, A21, A22;
6
7    FLAG_Part_2x2( A,  &ATL, &ATR,
8                       &ABL, &ABR,
9                       0,    0,   FLA_TL );
10
11   while ( FLAG_Obj_width( ATL ) < FLAG_Obj_width( A ) ){
12     b = min( min( FLA_Obj_width( ABR ),
13                   FLA_Obj_width( ABR ) ), nb_alg);
14
15     FLAG_Repart_2x2_to_3x3( ATL, ATR,   &A00, &A01, &A02,
16                                         &A10, &A11, &A12,
17                             ABL, ABR,   &A20, &A21, &A22,
18                                         b,    b,  FLA_BR );
19     /* ********************************************** */
20     FLAG_Chol_unb( A11 );
21
22     FLAG_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
23                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
24                FLA_ONE, A11, A21 );
25
26     FLAG_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
27                FLA_MINUS_ONE, A21, FLA_ONE, A22 );
28
29     /* ********************************************** */
30
31     FLAG_Cont_with_3x3_to_2x2( &ATL, &ATR, A00, A01, A02,
32                                            A10, A11, A12,
33                                &ABL, &ABR, A20, A21, A22,
34                                                  FLA_TL );
35   }
36 }
```

Fig. 2. `FLAG/C` implementation of the third algorithmic variant of the Cholesky factorization algorithm shown in Figure 3.

wrappers that encapsulate the indexing and storage details of the matrices, working with objects as defined above, and using a tuned implementation of the BLAS for the target graphics processor (CUBLAS [3]) as the underlying BLAS implementation.

This simple example shows how, by increasing the abstraction level, complex index manipulation can be avoided in the codes, reducing programming errors and increasing the reliability of the implementations.

III. A PRACTICAL ANALYSIS OF PERFORMANCE

In this section we evaluate the flexibility, ease-of-use, and efficiency of the interface using the Cholesky factorization.

A. `FLAME` *methodology: algorithmic variants*

Figure 3 shows all the unblocked and blocked algorithmic variants for the Cholesky factorization, obtained by applying the `FLAME` methodology. (The methodology details are out of the scope of the article, but a complete description can be found in [11].)

The (unblocked) variants on the left side of the figure are expressed in terms of scalar operations; the (blocked) variants on the right cast the bulk of the computation in terms of level-3 BLAS operations. On current processors, with several levels of cache memory, it is possible to carefully orchestrate the memory accesses for these type of operations achieving high performance. A few studies on modern GPUs [8], [7], [9] show how, for this type of operations, these hardware accelerators

can deliver up to $10\times$ speed-ups compared with highly tuned implementations on a general-purpose processor, even taking into account the overhead introduced by the data transfers through the PCI-Express bus.

B. Codification of variants with the FLAG/C API

Some important decisions have been adopted in the development of FLAG/C routines for the Cholesky factorization. First, note how all variants of the unblocked implementations of the Cholesky factorization shown in Figure 3 calculate the square root of a scalar value. Given the dependencies of the factorization process, it is not possible to group many square roots in a single SIMD operation. However, performing just one operation on the GPU can introduce an important overhead associated with the thread creation and the execution of a kernel without the necessary computational load per element. Therefore, all the implementations discussed next compute the square roots on the CPU. We propose two different implementations for each algorithmic variant of the Cholesky factorization algorithm shown in Figure 3.

1) First family of implementations: The first group of implementations, imp1, retrieve each diagonal element to CPU memory, compute the square root there, and then send the result back to GPU memory. All remaining operations of the factorization are computed in the GPU. The transfers of the diagonal elements have to be explicitly inserted in the code in Figure 2 by the programmer (using simple routines from the FLAG/C API). The existence of transfer routines is justified as a tradeoff between simplicity-of-use and efficiency.

The imp1 implementations have two main sources of inefficiency. First, if the block size n_b is small compared with the matrix size n, the factorization of the diagonal block performs many calls to level-1 and level-2 BLAS, with reduced computational load per call. For example, in Variant 3, $\frac{n}{n_b}(n_b-1)$ invocations of the scaling routine (FLAG_Inv_Scal in the API) with at most n_b-1 floating-point operations (flops) each are done, and the same number of symmetric rank-1 updates (FLAG_Syr in the API) with at most $\frac{n_b(n_b-1)}{2}$ flops each are performed. For that example, we can avoid the penalty associated with the reduced computational load per call by increasing the block size; unfortunately, this decision decreases the number operations performed in terms level-3 BLAS routines. Second, the imp1 implementations perform 2 transfers per matrix row/column, of size 4 bytes each (we assume real single-precision entries for the matrix). The cost of transferring a message of size t via the PCI-Express bus can be modeled as $\alpha+\beta t$ where α and β denote, respectively, the latency and bandwidth of the bus. Thus, as in general $\alpha \gg \beta$, it is important to design an alternative solution to group several tiny transfers into a larger message.

2) Second family of implementations: Taking into account the much higher cost of the latency, the second family of implementations, imp2, can be viewed as a generalization of those in imp1. In this case, the diagonal blocks of size n_b are transferred to the CPU memory, factored there, and the result is then sent back to the GPU memory. All other operations are performed on the GPU.

The total transfer cost for this family of implementations can be modeled as $T_2(n,n_b)=\frac{n}{n_b}(2\alpha+8\beta n_b^2)$, while this cost for the first family of implementations as $T_1(n)=n(2\alpha+8\beta)$. Thus, $T_2(n,n_b) \leq T_1(n)$ for any block sizes in the range $1 \leq n_b \leq max(1,\frac{\alpha}{4\beta})$. As usually $\alpha \gg \beta$, it is likely that $max(1,\frac{\alpha}{4\beta})>1$.

IV. Experimental results

In this section, we evaluate the efficiency of the FLAG/C API using several codes for the Cholesky factorization.

Performance results are reported in terms of MFLOPS, defined as the millions of flops per second. The number of flops is common to all the implementations, and can be modeled as $\frac{n^3}{3}$, where n the dimension of the matrix.

All measured times include the time required to initially transfer the whole matrix from CPU memory to GPU memory, and retrieve the Cholesky factor once the factorization is completed. For the blocked implementations, integer powers of 2 in the interval $[1,256]$ were evaluated for the block size, n_b. The hardware setup is shown in Table I. In our experiments, we employ Intel MKL 10.0 implementation of BLAS on a single core of the Intel processor, and CUBLAS 1.1 for the GPU operations.

	CPU	GPU
Processor	Intel Core 2 Duo	NVIDIA 8800 Ultra
Codename	Crusoe E6320	G80
Cores	2	128
Clock frequency	1.86 GHz	575 MHz
Memory frequency	2×333 MHz	2×900 MHz
Word width	64 bits	384 bits
Max. bandwidth	5.3 GB/s	86.4 GB/s
Memory type	1024 MB DDR2	768 MB GDDR3
I/O bus	PCI Express x16 (8 GB/s)	

TABLE I
HARDWARE SETUP USED IN THE EXPERIMENTAL PART.

To evaluate the overhead introduced by FLAG/C, we have implemented all variants using LAPACK-style codes [12] replacing all BLAS calls from within the LAPACK codes with CUBLAS invocations. Also, to compare the performances of the GPU with the CPU when executing FLAME codes, we have implemented all variants for the Cholesky factorization using the FLAME/C interface for CPUs.

Figures 4, 5, and 6 show, respectively, the performance attained on the GPU with the imp2 implementation of variants 1, 2, and 3 of the blocked Cholesky factorization. The left and right-hand sides of the figure show the performance, respectively, of the corresponding LAPACK-style and FLAG/C codes.

The results of this first experiment report that the overhead introduced by the FLAG/C interface is negligible. Indeed, with the scale used in the representations, it is not possible to distinguish any difference between the LAPACK-style and

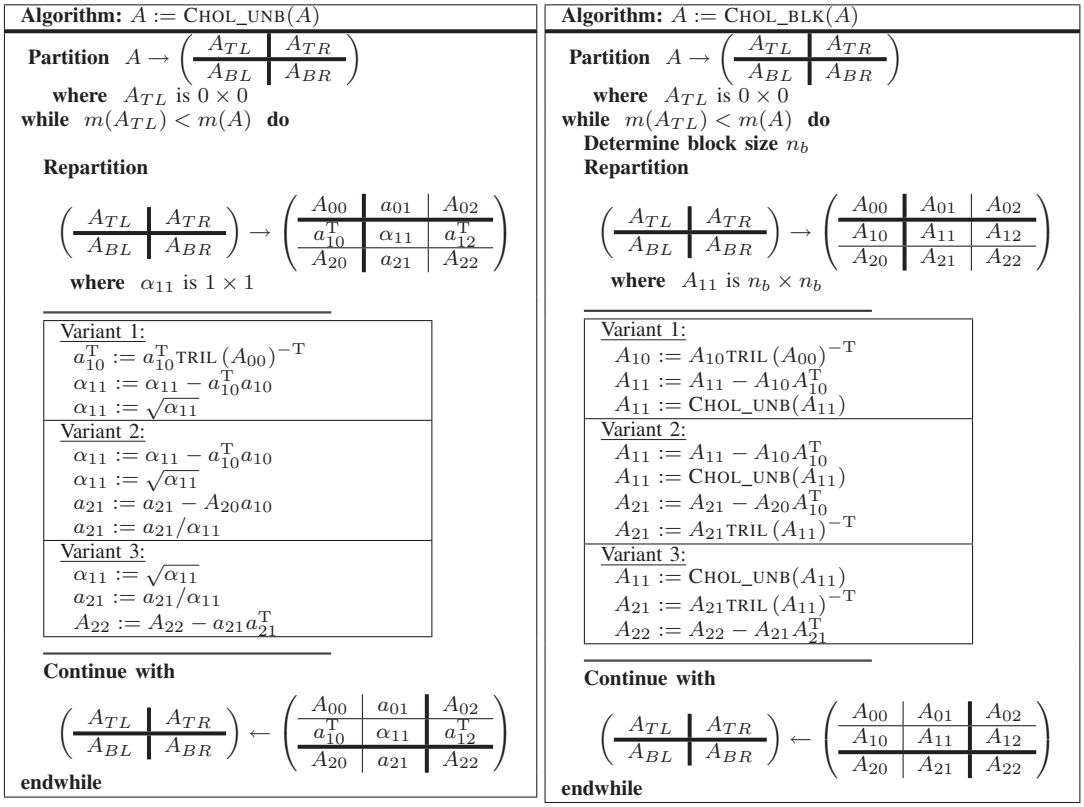

Fig. 3. Algorithmic variants of the Cholesky factorization represented using the FLAME notation. Left: scalar implementations. Right: blocked implementations.

the `FLAG/C` implementations. The results thus confirm that the simplification in the development process does not lead to a penalty in the performance attained by these routines.

The three variants show a similar behavior as a function of the block size. For large n, the performance increases with n_b up to a certain block size; from this point, the performance decreases. This threshold is reached when the gain introduced by invoking BLAS-3 routines with a larger block size is compensated by the penalty incurred by casting a larger part of the computational load in the factorization of the diagonal blocks. The range of block sizes evaluated for variant 1 are not big enough to reach this crossover point.

The correct choice of the algorithmic variant is critical to attain high performance. Variant 2 of the blocked implementation attains much higher performance than the other two. The difference is due to the specific Level-3 CUBLAS kernels involved in each variant. The analysis in [7] revealed that the kernel `GEMM` in CUBLAS 1.1 is highly tuned while others like `TRSM` or `SYRK` are not. The former kernel is precisely the routine which concentrates most of the computational load of variant 2. Note that the other two variants do not even invoke the `GEMM` kernel.

It is also notable the irregular behavior in the performance of all three versions (see, for example, the attained performance when $n = 4000$.) This fact has already been observed in previous work [7], [9], with a solution based on padding proposed there. Memory access patterns on the GPU have a dramatic impact on the final performance of the algorithms. Thus, it is usual that, for matrix sizes that are an integer multiple of 16, the performance can be boosted by the data access pattern [9].

Figure 7 reports the performance attained by the GPU executing both implementations of variant 3 of the blocked Cholesky factorization routine; we obtained similar results for the other two variants of the routine. Comparing the performance of both alternatives, it is possible to conclude that `imp2` clearly outperforms `imp1`, specially for small matrices. The difference in performance between both implementations progressively decreases with n. For a fixed block size (as is the case of the experiments of the Figure 7), the "weight" of factorizing the diagonal blocks compared with that of the full matrix factorization decreases as n is increased. In terms of the number of flops, this weight can be modeled as

$$\frac{\frac{n_b^3}{3} \times \frac{n}{n_b}}{\frac{n^3}{3}} = \frac{n_b^2}{n^2}, \tag{1}$$

which tends to zero when n is increased and n_b is fixed.

In order to justify with more detail the differences observed in Figure 7, we have measured the time required by the factorization of *all* diagonal blocks, and the time necessary to transfer these diagonal blocks from the GPU to the CPU and back. The first magnitude will be denoted by $F_1(n, n_b)$

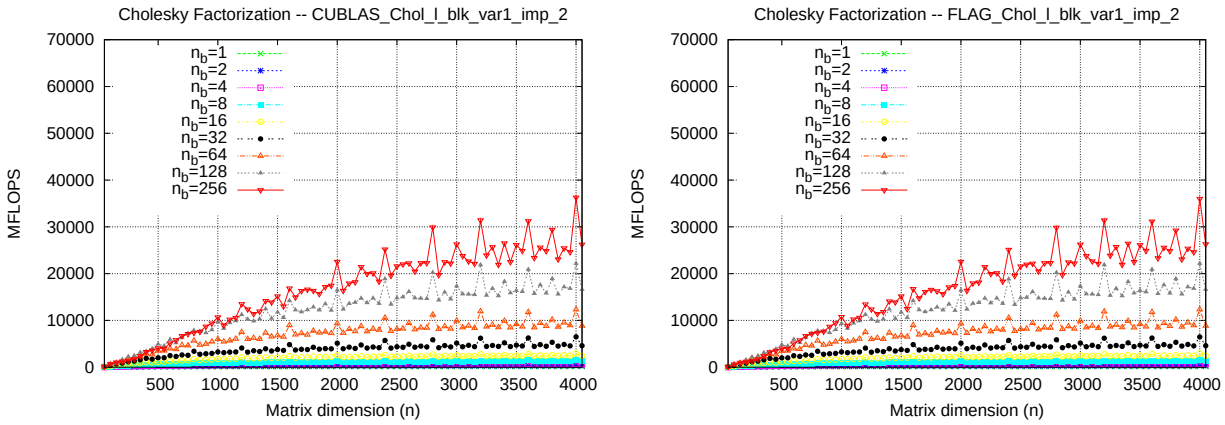

Fig. 4. Performance attained by the `imp2` implementations of Variant 1 for the Cholesky factorization on the GPU. Left: LAPACK-style code. Right: `FLAG/C` code

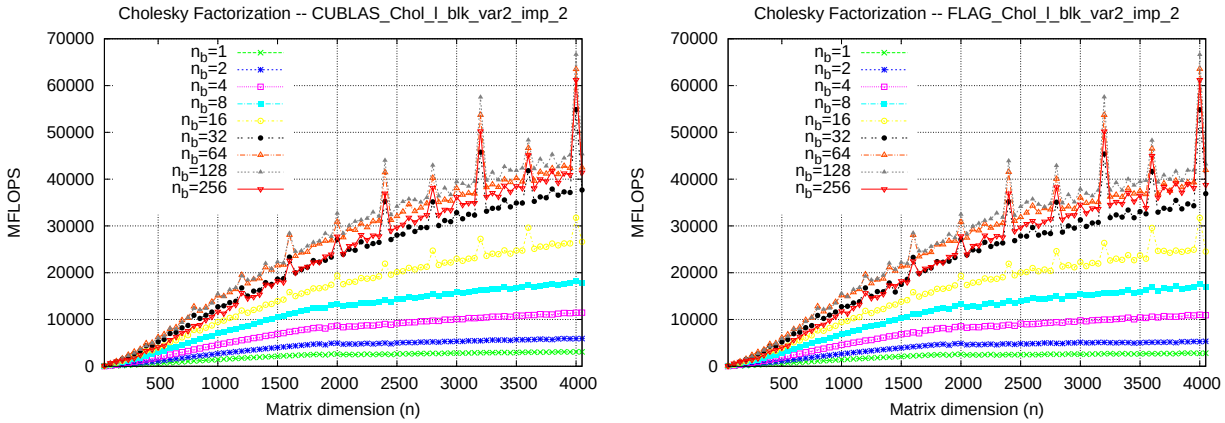

Fig. 5. Performance attained by the `imp2` implementations of Variant 2 for the Cholesky factorization on the GPU. Left: LAPACK-style code. Right: `FLAG/C` code

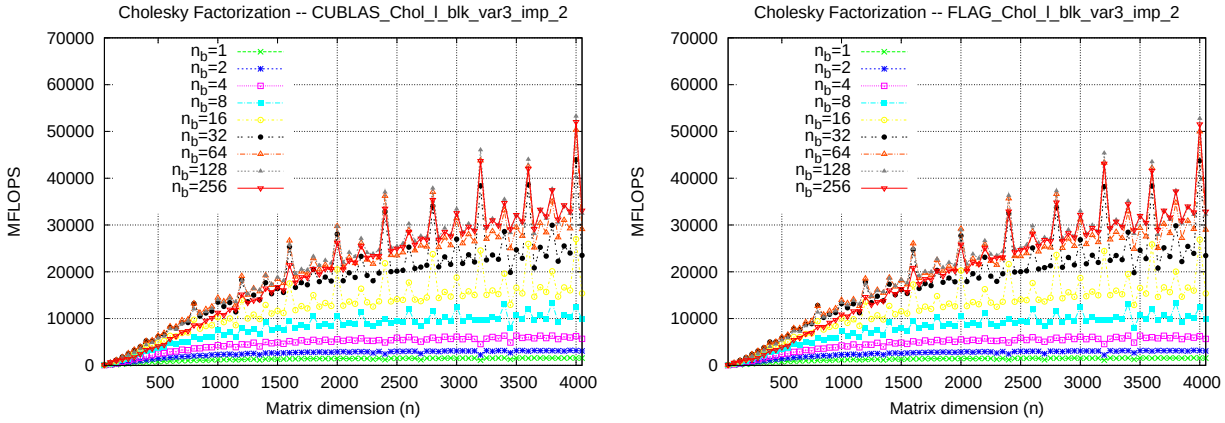

Fig. 6. Performance attained by the `imp2` implementations of Variant 3 for the Cholesky factorization on the GPU. Left: LAPACK-style code. Right: `FLAG/C` code

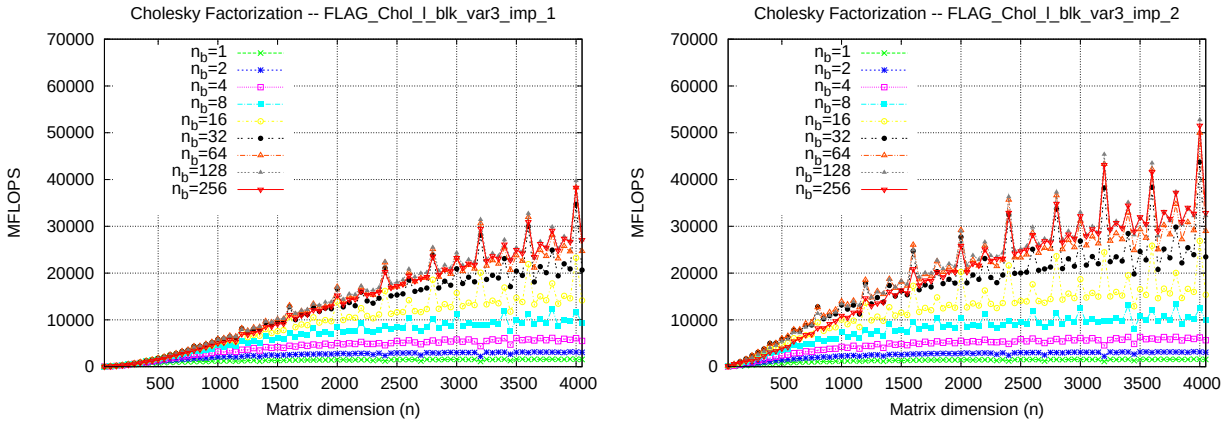

Fig. 7. Performance attained by `imp1` and `imp2` implementations of Variant 3 for the Cholesky factorization on the GPU. Left: `imp1` implementation. Right: `imp2` implementation

and $F_2(n, n_b)$ for the first and second implementations, respectively, while the second magnitude will be denoted by $T_1(n)$ and $T_2(n, n_b)$, respectively.

Figure 8 illustrates that $F_1(n, n_b) \gg F_2(n, n_b)$, in an experimental demonstration that it is not possible to amortize the overhead introduced by the initialization and start up of a large number of threads in the GPU for a low-cost linear algebra operation. This is precisely the case for the implementation `imp1` and the GPU operations involved in the factorization of the diagonal blocks. As in the implementation `imp1` there are progressively more CUBLAS invocations, the difference between both implementations progressively increases.

Another important observation that can be extracted from Figure 8 is that $T_2(n, n_b) < T_1(n)$, with block sizes $n_b > 1$. In addition, by increasing the block size, the difference between both values also gets larger until $n_b = 32$, when the balance between the overhead of transferring the diagonal blocks is compensated by the smaller latency.

We can estimate the parameters of the model by a least squares fitting of the curve $1000\alpha\frac{1}{n_b} + 4000\beta n_b$ to the measurements of $T_2(500, n_b)$; see Figure 9. The results there show the accuracy of this fitting; in addition, it confirms that the transfer cost of a message through the PCI-Express bus matches the model $\alpha + \beta n$, with $\alpha = 1.5968e^{-5}$ sec., and $\beta = 2.9715e^{-9}$ sec./byte. Thus, $max(1, \frac{\alpha}{4\beta} \approx 1343) \gg 1$, value for which the model estimates that $T_1(n) \approx T_2(n, n_b)$. The asymptotic bandwidth, that is, the observed bandwidth for those transfers with message size large enough to ignore the latency is $\frac{8}{\beta \times 10^9} = 2.69$ Gbit/sec. For both implementations of the `CHOL_BLK` algorithm, the number of messages and the amount of transferred data is the same in both ways (from and to GPU); in addition, given the limitations of CUBLAS 1.1, it is not possible to overlap data transfers and calculation on the GPU. Thus, the upper bound for the real peak bandwith will be a 50% (4 GByte/sec.) of the maximum theoretical bandwith (8 GByte/sec.). However, our experimental study offers a real bandwidth which is only 8% of the former upper bound.

Finally, Figure 10 shows the performance of all three

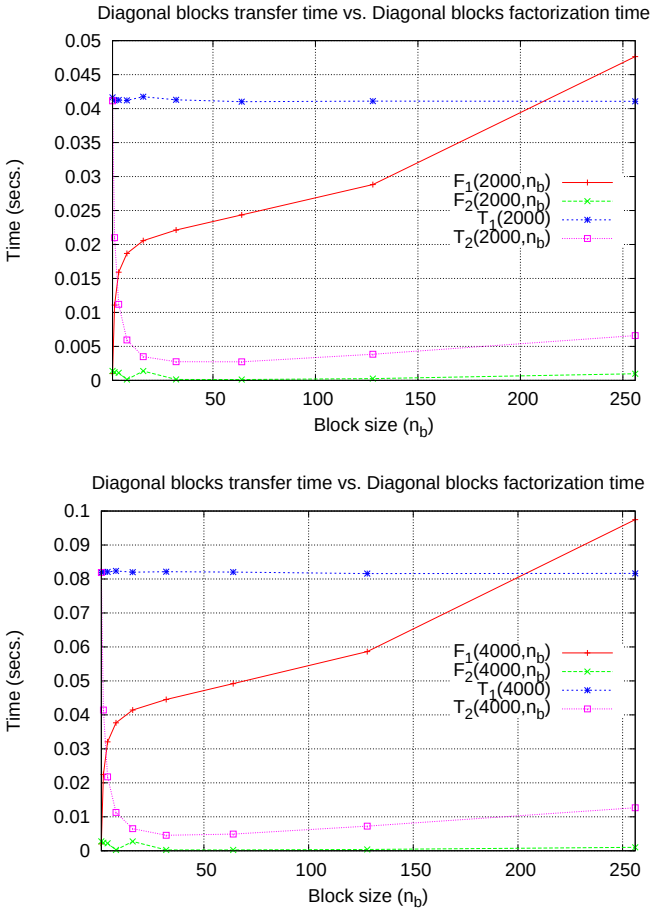

Fig. 8. Time for the factorization of all diagonal blocks $(F_1(n, n_b), F_2(n, n_b))$ and time required to transfer all these blocks $(T_1(n), T_2(n, n_b))$ between the CPU and GPU memory spaces for the implementations of the blocked Cholesky factorization. Top: $n = 2000$. Bottom: $n = 4000$.

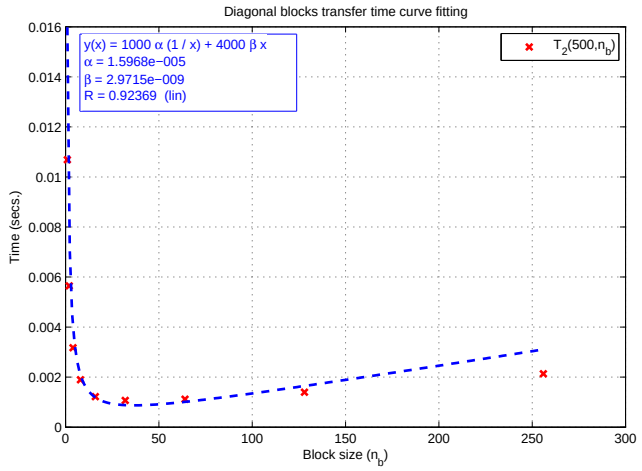

Fig. 9. Least squares fitting of the theoretical curve $1000\alpha\frac{1}{n_b} + 4000\beta n_b$ and the experimental results for $T_2(500, n_b)$

variants of the blocked Cholesky factorization routine on the CPU implemented using the `FLAME/C` interface. Comparing the results in this figure with those in Figures 4, 5, and 6 it is possible to conclude that the GPU can accelerate up to 7 times the Cholesky factorization for this hardware configuration.

V. Conclusions

We have described a new API which facilitates the development of dense linear algebra codes on graphics processors, while extracting much of the high-performance of these architectures. Using the FLAME notation, the proposed library provides a wide-appeal tool to develop optimized codes for dense linear algebra with little effort.

The use of this library has been illustrated using the Cholesky factorization of a dense matrix, an operation of interest in the solution of certain dense linear systems. The analysis of several variants and implementations revealed some useful insights and optimization techniques for the G80 architecture. Much of these results also apply to other major routines key to the solution of linear systems and linear-least squares problems such as the LU and QR factorizations.

Acknowledgments

This work has been supported by the Spanish Office of Education and Science through the project CICYT TIN2005-09037-C02-02 and FEDER, and the project P1-1B2007-32 of the Fundación Caixa Castelló/Bancaixa and the Universidad Jaume I.

References

[1] NVIDIA Corp.: CUDA Programming Guide, version 2.0, NVIDIA Corp., USA, 2008.
[2] AMD: Brook+, SC07 BOF Session, November 13, 2007
[3] NVIDIA Corp.: CUBLAS Library, NVIDIA Corp., USA, 2008.
[4] ACML-GPU, http://ati.amd.com/technology/streamcomputing/sdkdwnld.html. ltima visita, 2 de octubre de 2008.
[5] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra, MPI-The Complete Reference, MIT Press, Cambridge, MA, USA

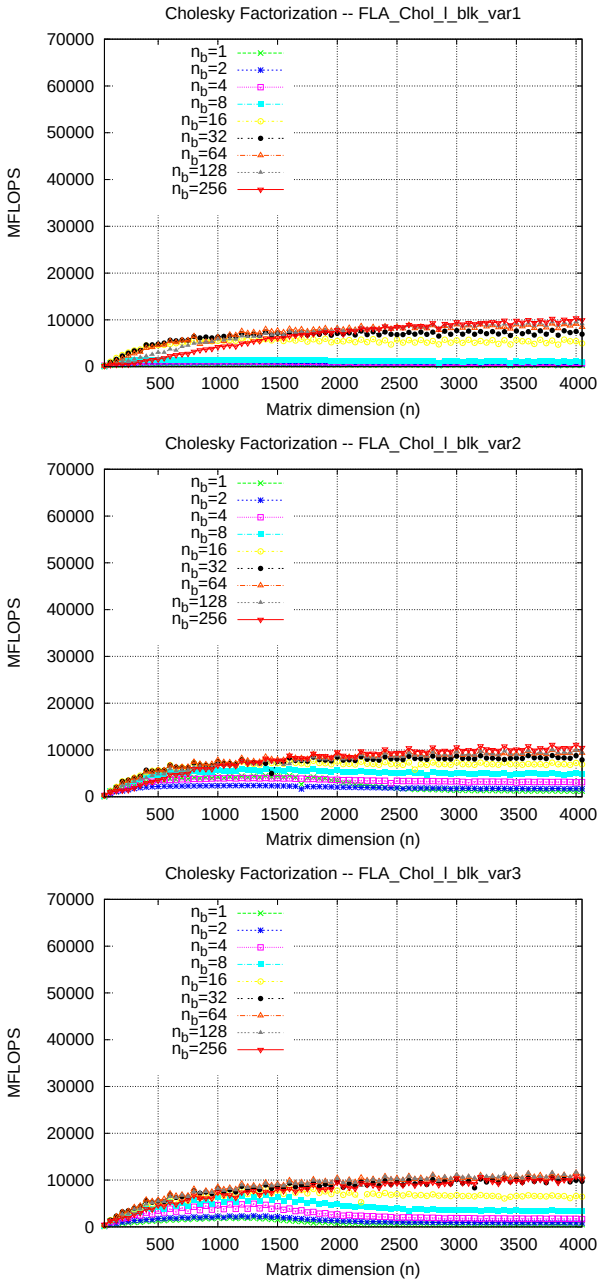

Fig. 10. Performance attained by `imp2` implementations for the Cholesky factorization on the CPU.

[6] S. Balay, W.D. Gropp, L.C. McInnes, B.B. Smith, PETSc 2.0 users manual. Technical Report ANL-95/11, Argonne National Laboratory, Apr. 1999
[7] S. Barrachina, M. Castillo, Francisco D. Igual, R. Mayo, E. S. Quintana-Ort, Evaluation and tuning of the level 3 CUBLAS for graphics processors, in: Workshop on Parallel and Distributed Scientific and Engineering Computing, PDSEC 2008(CD-ROM). Miami (EE.UU.). 2008.
[8] O. Schenk, M. Christen, H. Burkhart, Algorithmic Performance Studies on Graphics Processing Units, J. Parallel Distrib. Comput. 68 (2008) 1360-1369.
[9] S. Barrachina, M. Castillo, F. Igual, R. Mayo, E. S. Quintana, Solving dense linear systems on graphics processors, Lecture Notes in Computer Science 5168, Euro-Par 2008,(Eds. E. Luque, T. Margalef, D. Bentez,) pp. 739-748. Las Palmas de Gran Canaria (Espaa). 2008
[10] R. A. van de Geijn, Using PLAPACK: Parallel Linear Algebra Package,

The MIT Press, 1997
[11] R. A. van de Geijn, E. S. Quintana-Ortí, The Science of Programming Matrix Computations, www.lulu.com, 2008
[12] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, Society for Industrial and Applied Mathematics, 1999
[13] G. H. Golub, C. F. Van Loan, Matrix Computations, 3rd edition, The Johns Hopkins University Press, Baltimore, 1996
[14] P. Bientinesi, E. S. Quintana-Ortí, R. A. van de Geijn, Representing Linear Algebra Algorithms in Code: The FLAME Application Programming Interfaces, ACM Trans. Math. Soft., volume 31, pages 27–59, 2005
[15] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, R. A. van de Geijn, The Science of Deriving Dense Linear Algebra Algorithms, ACM Transactions on Mathematical Software, volume 31, pages 1–26, 2005