Retargeting PLAPACK to Clusters with Hardware Accelerators

FLAME Working Note #42

Manuel Fogué* Francisco D. Igual* Enrique S. Quintana-Ortí*
Robert van de Geijn†

February 11, 2010

Abstract

Hardware accelerators are becoming a highly appealing approach to boost the raw performance as well as the price-performance and power-performance ratios of current clusters. In this paper we present a strategy to retarget PLAPACK, a library initially designed for clusters of nodes equipped with general-purpose processors and a single address space per node, to clusters equipped with graphics processors (GPUs). In our approach data are kept in the device memory and only retrieved to main memory when they have to be communicated to a different node. Here we benefit from the object-based orientation of PLAPACK which allows all communication between host and device to be embedded within a pair of routines, providing a clean abstraction that enables an efficient and direct port of all the contents of the library. Our experiments in a cluster consisting of 16 nodes with two NVIDIA Quadro FX5800 GPUs each show the performance of our approach.

1 Introduction

Dense linear algebra operations lie at the heart of many scientific and engineering applications. The large dimensions of many of the problems arising in these applications, and the cubic computational cost of the algorithms for dense linear algebra operations, led to the development of high performance libraries like LAPACK [1] for single processors and shared-memory platforms, and ScaLAPACK [4] for distributed-memory (i.e., message-passing) parallel architectures. LAPACK and ScaLAPACK are written in Fortran-77, with little concessions to high-level abstractions and much less to object-oriented programming. On the other hand, our "object-based" alternative library `libflame` [8], for multi-core processors and in general shared-memory platforms, encapsulates information like data layout in memory, avoiding the error-prone manipulation of indices. PLAPACK [7] (which inspired `libflame`) further exercises this object-based approach to also abstract the user from the distribution of the data among the memory spaces of a distributed-memory platform.

While existing libraries like ScaLAPACK and PLAPACK provide efficient codes for the solution of large-scale dense linear algebra operations on clusters of computers, the incorporation of hardware accelerators in these platforms appears as a new challenge for library developers. Hardware accelerators are increasingly becoming a cheap alternative to accelerate applications that are intensive in floating-point arithmetic. Sony+Toshiba+IBM Cell B.E., NVIDIA and AMD/ATI GPUs, ClearSpeed ASICs and FPGAs from different vendors are all targeting the high performance computing market with different approaches, from the heterogeneous Cell to the many-core GPUs. The RoadRunner supercomputer, ranked #2 in the November 2010 Top500 list (`http://www.top500.org`), employs this accelerator technology to deliver more than 1 PFLOPS (1 PFLOPS = 10^{15} floating-point arithmetic operations –*flops*– per second) and almost 445 MFLOPS/W [5].

*Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain. {`figual,quintana`}`@icc.uji.es`.

†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712. `rvdg@cs.utexas.edu`

It is thus natural that many other platforms, from medium-sized clusters to supercomputers, will follow this trend in the near future.

Programming dense linear algebra algorithms for message-passing parallel architectures is a task for experts. The complexity arises from the existence of multiple separate address spaces (one per node in the cluster), the need to synchronize the processes collaborating in the computation so as to minimize idle times, and the importance of reducing the amount of communication as well as hiding message-passing latency. The use of hardware accelerators to build *hybrid clusters* may exacerbate the complexity of programming these platforms, by introducing a new separate memory space in the accelerator (or the device) different from that of the host (i.e., the main memory of the node). How to easily "rewrite" message-passing dense linear algebra libraries, like ScaLAPACK and PLAPACK, to deal with hybrid clusters is the question we address in this paper.

A straight-forward approach to retarget codes from existing dense linear algebra libraries to hybrid clusters is to hide the presence of the hardware accelerators inside the routines that perform the "local" computations. In this approach matrices are kept in the host memory most of the time. Then, when a computation is off-loaded to the device, only the data that are strictly necessary are transferred there, exactly at that moment, and the results are retrieved back to the host memory immediately after the computation is completed. Data transfers between the memories of host and device can thus be easily encapsulated within wrappers to the BLAS and LAPACK routines that perform local computations. This is partly done implicitly in ClearSpeed accelerators: Routines from ClearSpeed implementation of the matrix-matrix product and a few other basic linear algebra operations assume that data are stored in the host memory and perform the transfers transparently to the user.

The invocation of kernels from NVIDIA CUBLAS, on the other hand, requires explicit handling of the transfers, though developing wrappers for this library is relatively simple. The major advantage of this approach is that no change other than developing these simple wrappers is necessary to retarget the library to a hybrid cluster. In other words, this approach leads to an almost transparent port of most of the contents of ScaLAPACK and PLAPACK to hybrid clusters. While straight-forward, the approach can incur a nonnegiblible number of data transfers between the memories of the host and the device, degrading performance. For example, if the same chunk of data is involved in two consecutive local computations, its transfer between host and device is unnecessarily repeated. To tackle this problem, our approach to port dense linear algebra codes to hybrid clusters places the data in the accelerators memory most of the time (this has been independently done in [5] for a cluster equipped with Cell B.E. accelerators). A data chunk is recovered to the host memory only when it is to be sent to a different node, or when it is involved in an operation that is to be computed by the host. Following PLAPACK object-based approach, all these data movements are encapsulated within PLAPACK copy and reduce communication operations, thus leading also to an almost transparent port of the contents of this library to hybrid clusters.

The paper is structured as follows. In Section 2 we employ the Cholesky factorization to briefly overview the high-level approach to coding dense linear algebra operations intrinsic to PLAPACK. In Section 3 we describe how to make an almost transparent retarget of PLAPACK, while attaining high-performance. In Section 4 we illustrate the benefits of this approach, providing experimental results on a 16-node cluster equipped with NVIDIA Quadro FX5800 GPUs. Section 5 summarizes the concluding remarks.

2 Computing the Cholesky Factorization in PLAPACK

The Cholesky factorization is usually computed as the first step in solving the linear system $Ax = b$, where $A \in \mathbb{R}^{n\times n}$ is a (dense) symmetric positive definite (SPD) matrix. This factorization is defined as

$$A = LL^T, \tag{1}$$

where $L \in \mathbb{R}^{n\times n}$ is a lower triangular matrix known as the Cholesky factor of A.

The PLAPACK blocked algorithm for the Cholesky factorization is given in routine `PLA_Chol` in Figure 1. (We have modified this code slightly to hide irrelevant details for the following discussion.) The code sample illustrates the object-based approach of PLAPACK: all data are encapsulated in `PLA_Obj` objects

```
 1  int PLA_Chol( int nb_alg, PLA_Obj A )
 2  {
 3    PLA_Obj  ABR = NULL,
 4             A11 = NULL,     A21 = NULL;
 5    /* ... */
 6
 7    /* View ABR = A */
 8    PLA_Obj_view_all( A, &ABR );
 9
10    while ( TRUE ) {
11      /* Partition  ABR = / A11  ||   *  \
12       *                 | =====  ===== |
13       *                 \ A21  || ABR  /
14       * where A11 is nb_alg x nb_alg      */
15      PLA_Obj_split_4( ABR, nb_alg, nb_alg,
16                           &A11, PLA_DUMMY,
17                           &A21, &ABR );
18
19      /* A11 := L11 = Cholesky Factor( A11 ) */
20      PLA_Local_chol( PLA_LOWER_TRIANGULAR, A11 );
21
22      /* Update A21 := L21 = A21 * inv( L11' ) */
23      PLA_Trsm( PLA_SIDE_RIGHT, PLA_LOWER_TRIANGULAR,
24                PLA_TRANSPOSE,  PLA_NONUNIT_DIAG,
25                one, A11,
26                     A21 );
27
28      /* Update A22 := A22 - L21 * L21' */
29      PLA_Syrk( PLA_LOWER_TRIANGULAR, PLA_NO_TRANS,
30                minus_one, A21,
31                one,       ABR );
32    }
33    /* ... */
34  }
```

Figure 1: (Simplified version of) PLAPACK right-looking blocked algorithm for computing the Cholesky factorization.

and accessed only via PLAPACK routines `PLA_Obj_view_all`, `PLA_Obj_split_4`, `PLA_Local_chol`, `PLA_Trsm`, and `PLA_Syrk`.

Following common practice, the factorization algorithm in Figure 1 overwrites the lower triangular part of the array A with the entries of L, while the strictly upper triangular part of the matrix remains unmodified. The code proceeds as follows. Initially, `ABR` is created as a view to `A` (line 8). (A view in PLAPACK is an object that references the same data as the parent object or view from which it was derived. Thus, `A` and `ABR` both reference the same data.) The `while` loop then iterates to split `ABR` into four quadrants (lines 15-17), as in

$$A_{BR} \rightarrow \left(\begin{array}{c|c} A_{11} & \star \\ \hline A_{21} & A_{22} \end{array} \right),$$

where the algorithmic block size `b` is assumed to match the distribution block size, so that of A_{11} resides on a single processor, and A_{BR} is reused to hold A_{22}. The computations in the loop body correspond to the right-looking variant of the Cholesky factorization: A_{11} is first decomposed as $A_{11} = L_{11}L_{11}^T$ (line 20), and the remaining blocks are updated as $A_{21} := A_{21}L_{11}^{-1}$ (lines 23-26) and $A_{22} := A_{22} - A_{21}A_{21}^T$ (lines 29-31) via, respectively, a triangular system solve (with multiple right-hand sides) and a symmetric rank-`nb_alg` update.

3 PLAPACK for Hybrid Clusters

We describe in this section our approach to retarget PLAPACK to a cluster equipped with hardware accelerators. To reduce the number of data transfers between the memory spaces of host and device, we propose to keep all data in the device memory most of the time. Given that current NVIDIA accelerator boards include a RAM of 4 GBytes, we do not expect that the size of the device memory becomes a limiting factor for most applications. Otherwise, one could still handle the device memory as a cache of the host memory, by implementing a software coherence protocol as, e.g., in [6].

3.1 Communicating data in PLAPACK

The primary vehicles for communication in PLAPACK are the copy and reduce operations. The approach in this library is to describe the distribution for the input and output using linear algebra objects, and then to copy or to reduce from one to the other.

Thus, a prototypical communication is given by the call `PLA_Copy( A, B )` where included in the descriptors `A` and `B` is the information for the respective distributions. Inside of the `PLA_Copy` routine it is determined how data must be packed, what collective communication must be called to redistribute, and how the data must be unpacked after the communication. What this means is that, with the addition of memory local to an accelerator, the necessary data movement with the host processors that perform the communication needs to be added. This can be accomplished by hiding the details completely within the `PLA_Copy` routine. The `PLA_Reduce` routine similarly allows contributions from different processors to be consolidated.

3.2 Data movement in the Cholesky factorization

Let us focus on the call `PLA_Trsm( ..., A11, A21 )`. In the particular implementation of the Cholesky factorization given in Figure 1, to start, `A11` exists within one processor and `A21` within one column of processors, with all elements of a given row of `A21` assigned to the same processor. Inside this routine, `A11` is broadcast so that all processors that own part of `A21` receive a copy, after which local triangular solves complete the desired computation. The call to `PLA_Syrk` performs similar data movements and local computations.

We note that PLAPACK includes more complex implementations that require more complex data movements. We purposely focus on a simple implementation since it captures the fundamental issues.

3.3 Changes to PLAPACK

In this section we describe how porting to an exotic architecture, like accelerators with local memories, is facilitated by a well-layered, object-based library like PLAPACK. We do so by exposing a small sampling of the low-level code in PLAPACK and discussing how this code had to be changed. We stress that the changes were made by the coauthor on this paper who had no experience with PLAPACK without any direct or indirect help from the coauthor who wrote PLAPACK.

We will employ the copy from a PLAPACK object of matrix type to a second object, of the same type, in order to illustrate the modular structure of PLAPACK and the series of changes that had to be made to accommodate the use of accelerators in PLAPACK. This copy is implemented in routine `PLA_Copy_from_matrix_to_matrix`, which we transformed into `CUPLA_Copy_from_matrix_to_matrix`.

Several cases are treated within the copy routine. For example, when both matrices are aligned to the same template (i.e., the contents are distributed among the nodes following the same pattern), local copies from the buffer containing the elements of the source matrix to the one with those of the target matrix suffice, as shown in the following excerpt of PLAPACK code:

```
/* PLAPACK PLA_Copy between aligned matrices */
if ( align_col_from == align_col_to ) {
  if ( align_row_from == align_row_to ){
    PLA_Local_copy( Obj_from, Obj_to );
    done = TRUE;
  }
}
```

In PLAPACK, the local copy inside `PLA_Local_copy` is performed using (several invocations to) the BLAS-1 routine `scopy`; in CUPLAPACK we call the analogous copy counterpart from the CUDA runtime to move data between different positions of the device memory.

A more elaborate case occurs when the two matrices feature different alignments. For simplicity, consider that the matrices share the same column alignment but differ in the row alignment. Thus, each column process has to send the corresponding block to each one of the processes in the same column. For efficiency, before these blocks are sent, they are packed in an auxiliary buffer to hide part of the communication latency by increasing the granularity of messages. This is done in PLAPACK as follows:

```
/* PLAPACK PLA_Copy between column aligned matrices */
while ( TRUE ){
  PLA_Obj_split_size( from_cur, PLA_SIDE_TOP,
                      &size_from, &owner_from );
  PLA_Obj_split_size( to_cur,   PLA_SIDE_TOP,
                      &size_to,   &owner_to);

  if ( 0 == ( size = min( size_from, size_to ) ) )
    break;

  PLA_Obj_horz_split_2( from_cur, size,
                        &from_1, &from_cur );
  PLA_Obj_horz_split_2( to_cur,   size,
                        &to_1,   &to_cur );

  if ( myrow == owner_from &&
       owner_from == owner_to ){
    PLA_Local_copy( from_1, to_1 );
  }
  else{
    if ( myrow == owner_from ) {
      PLA_Obj_get_local_contents(
          from_1, PLA_NO_TRANS,
          &dummy, &dummy,
          buffer_temp, size, 1 );

      MPI_Send(
          BF( buffer_temp ),
          size * local_width, datatype,
          owner_to, 0, comm_col );
    }
    if ( myrow == owner_to ) {
      MPI_Recv(
          BF( buffer_temp ),
          size * local_width, datatype,
          owner_from, MPI_ANY_TAG, comm_col,
          &status );

      PLA_Obj_set_local_contents(
          PLA_NO_TRANS, size, local_width,
          buffer_temp, size, 1, to_1 );
    }
  }
}
PLA_free( buffer_temp );
```

In PLAPACK, routine `PLA_Obj_get_local_contents` copies the local contents of the object into a buffer:

```
/* PLAPACK PLA_Obj_get_local_contents */

int PLA_Obj_get_local_contents   (
        PLA_Obj    obj,          int    trans,
        int        *rows_in_buf, int    *cols_in_buf,
        void       *buf,         int    leading_dim_buf,
        int        stride_buf)
/* ... */
  for ( j=0; j<n; j++ ){
    tempp_local = buf_local + j*leading_dim_buf*typesize;
    tempp_obj   = buf_obj   + j*ldim*typesize;
    memcpy( tempp_local, tempp_obj, m*typesize );
  }
/* ... */
```

An equivalent effect is achieved in CUPLAPACK with a simple invocation to CUBLAS routine `cublasGetMatrix`, which packs the data into a contiguous buffer while simultaneously retrieving it from the device memory:

```
/* CUPLAPACK PLA_Obj_get_local_contents */

int CUPLA_Obj_get_local_contents   (
        PLA_Obj    obj,          int    trans,
        int        *rows_in_buf, int    *cols_in_buf,
        void       *buf,         int    leading_dim_buf,
        int        stride_buf)
/* ... */
      cublasGetMatrix( m, n, typesize,
                       buf_obj,   ldim,
```

Table 1: Detailed features of the Longhorn cluster. Peak performance data only consider the raw performance delivered by the CPUs in the cluster, excluding the GPUs.

	Per Node	Per System (256 nodes)
Number of cores	8	2,048
CPU	Intel Xeon Nehalem @ 2.53 GHz	
Available memory	48 Gbytes	13.5 TBytes
Interconnection network	QDR Infiniband	
Graphics system	128 NVIDIA Quadro Plex S4s	
GPU	2 x NVIDIA Quadro FX5800	512 x NVIDIA Quadro FX5800
Interconnection bus	PCI Express 2.0 (8x)	
Available video (device) memory	8 Gbytes DDR3	2 TBytes DDR3
Peak performance (SP)	161.6 GFLOPS	41.40 TFLOPS
Peak performance (DP)	80.8 GFLOPS	20.70 TFLOPS

```
                          buf_local , leading_dim_buf );
  /* ... */
```

Analogous changes are needed in the remaining cases of `PLA_Copy` and `PLA_Reduce`, the other key routine to data duplication and consolidation in PLAPACK, which together embed all data communication (transfer) in PLAPACK.

4 Experimental Results

The goal of the experiments in this section is twofold. First, to report the raw performance that can be attained with the accelerated version of the PLAPACK library for two common linear algebra operations: the matrix-matrix multiplication and the Cholesky factorization. Second, to illustrate the scalability of the proposed solution by executing the tests on a moderate number of GPUs.

Longhorn is a hybrid CPU/GPU cluster designed for remote visualization and data analysis. The system consists of 256 dual-socket nodes, with a total of 2,048 compute cores (Nehalem quad-core), 512 GPUs (128 NVIDIA Quadro Plex S4s, each containing 4 NVIDIA FX5800), 13.5 TBytes of distributed memory and a 210 TBytes global file system. The detailed specifications of the cluster can be found in Table 1.

On the software side, MKL 10.2 was employed for all computations performed on the Intel cores; NVIDIA CUBLAS (version 2.2) built on top of the CUDA application programming interface (version 2.2) provided the necessary linear algebra kernels for the GPUs; PLAPACK (release 3.2) was the base for our experiments and development of the GPU-accelerated codes; and MVAPICH2 (version 1.4) was the implementation of MPI. Single-precision arithmetic was employed in all experiments as the GPU is not competitive in double precision, due to the much smaller number of cores dedicated to this and the lack of an optimized double-precision implementation of BLAS. At this point, it is worth noting that refinement from single to double precision in the context of linear systems is well understood and has been reported as a practical approach for the solution of dense linear systems on GPUs in [2, 3]. Furthermore, Fermi (NVIDIA next generation of GPUs) promises to decrease the single- to double-precision gap in performance from the current factor of 8 to one of 2.

In our experiments we only employ 16 compute nodes from Longhorn. The results for 1, 2, 4, 8 and 16 GPUs make use of one of the GPUs in each node, with one MPI process per computing node. The results on 32 GPUs (there are two GPUs per node) were obtained using two MPI processes per node. This setup also illustrates the capability of the accelerated version of PLAPACK to deal with systems equipped with more than one accelerator per node (e.g., in a configuration with nodes connected to Tesla S1070 servers).

The matrix-matrix multiplication is frequently abused as a showcase of the highest attainable performance of the target architecture. Following this trend, we have developed an implementation of this operation based on the PLAPACK `PLA_Gemm` routine. The performance results for the matrix-matrix multiplication are shown in Figure 2 (left plot). The highest performance attained for the matrix multiplication is slightly over 8 TFLOPS when using 32 GPUs for matrices of dimension $61,440 \times 61,440$.

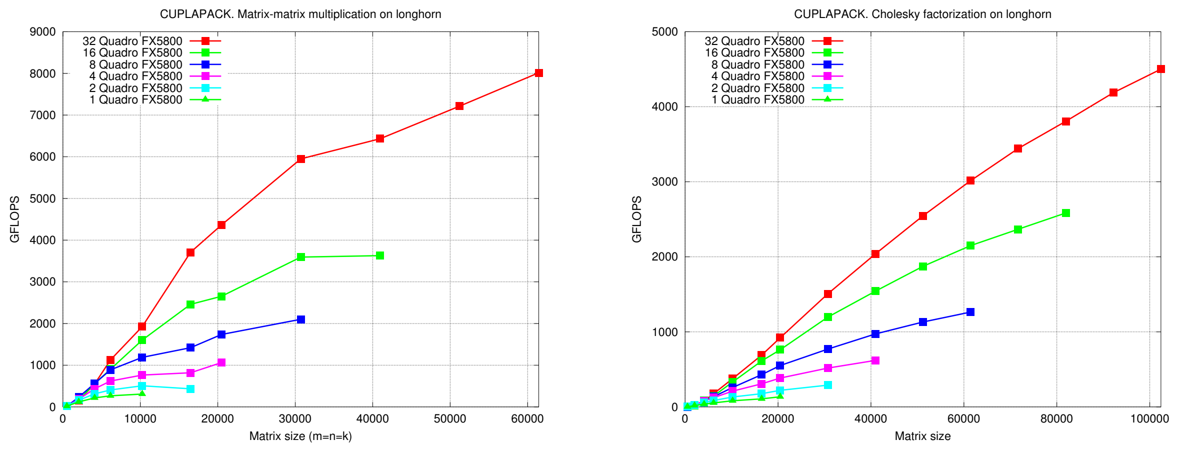


Figure 2: Performance of the PLAPACK-based accelerated codes for the matrix-matrix multiplication on Longhorn.

A similar experiment has been carried out for the Cholesky factorization. In our implementation of the PLAPACK routine `PLA_Chol`, the factorization of the diagonal blocks (routine `PLA_Local_Chol` in Figure 1) is computed in the CPU using the general-purpose cores and all remaining operations are performed on the GPUs. This hybrid strategy has been successfully applied in previous studies [2, 3, 6]. Figure 2 (right plot) reports the performance of the Cholesky routine on Longhorn, which delivers 4.4 TFLOPS for matrices of dimension $102,400 \times 102,400$.

A quick comparison between the top performance of the matrix-matrix multiplication using the accelerated version of PLAPACK ($\approx$ 8 TFLOPS) and the (theoretical) peak performance of the CPUs of the entire system (41.40 TFLOPS in single-precision arithmetic) reveals the advantages of exploiting the capabilities of the GPUs: using only a 6% of the graphics processors available in the cluster (32 out of 512 GPUs) it is possible to attain 20% of the peak performance of the machine if one used exclusively the available CPUs.

Figure 3 illustrates the scalability of the PLAPACK-based matrix-matrix multiplication routine: compared with the performance of the "serial" tuned implementation of the matrix-matrix product routine in CUBLAS, our routine achieves a speedup 22x is achieved on 32 GPUs, which demonstrates the scalability of the solution. Two main reasons account for the performance penalty: the Infiniband interconnect and the PCI Express bus (specially when 32 GPUs/16 nodes are employed as the bus is then shared by the two GPUs in each node).

The plots in Figure 4 evaluate the performance of the original PLAPACK implementation and the GPU-accelerated version of the library for the matrix-matrix multiplication (left plot) and the Cholesky implementation (right plot). Results are only shown on 16 nodes of Longhorn. Thus, the plots report the performance of the PLAPACK implementation using 128 Intel Nehalem cores (8 cores per node) versus those of the accelerated library using 16 and 32 GPUs (that is, one or two GPUs per each node). For the matrix-matrix multiplication, the highest performance for PLAPACK is 2.3 TFLOPS, while the accelerated version of the library attains 3.6 TFLOPS for 16 GPUs and 6.4 TFLOPS for 32 GPUs. Thus, the speedups obtained by the accelerated routines are 1.56x and 2.78x, respectively. For the Cholesky implementation, the PLAPACK routine attains a peak performance of 1.6 TFLOPS, compared with the 2.6 and 4.5 TFLOPS achieved by the accelerated versions of the routines on 16 and 32 GPUs, respectively. In this case, the speedups are 1.66x and 2.81x, respectively.

The benefits of using an approach in which data is kept in the GPU memory during the whole computation are shown in Figure 5. The results report the performance of the "simple" approach, in which data is stored

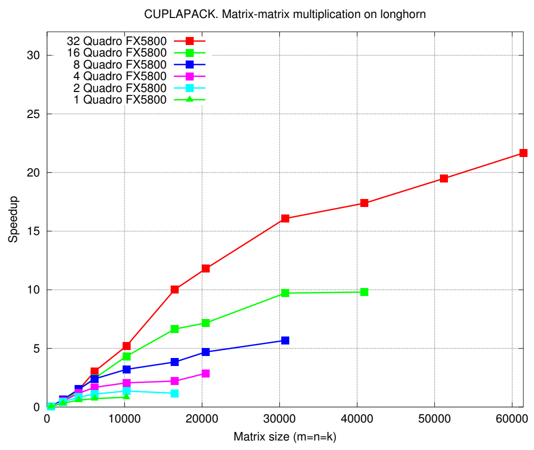


Figure 3: Scalability of the PLAPACK-based codes for the matrix-matrix multiplication on LONGHORN. The reference for the scalability results is the performance of CUBLAS SGEMM on one GPU, provided it could deal with matrices of the tested dimensions.

in the main memory of each node and transferred to the GPU when it is strictly necessary, and the "tuned" approach, in which data is kept (most of the time) in the GPU memory. The benefits of the second approach, in terms of higher performance, are clear, especially for large matrices. On the other hand, in the simple approach the size of the problem that can be solved is restricted by the amount of main memory in the system, which is usually larger than the available video memory (see the tested sizes for the matrix-matrix multiplication in Figure 5). In principle, this can be solved transparently to the programmer in the tuned approach, by handling the device memory as a cache of the host memory [6].

5 Concluding Remarks

We have presented an approach to mechanically port the routines of the dense linear algebra message-passing library PLAPACK to a hybrid cluster consisting of nodes equipped with hardware accelerators. By initially placing all data in the memory of the accelerators, the number of PCI Express transfers between the memories of the host and the device is reduced and performance is increased. All data transfers are embedded inside PLAPACK communication (copy) and consolidation (reduce) routines so that the retarget of the library routines is mostly automatic and transparent to the user.

The experimental results have demonstrated that the integration of GPUs in the nodes of a cluster is an efficient, cheap and scalable solution for the acceleration of large dense linear algebra problems.

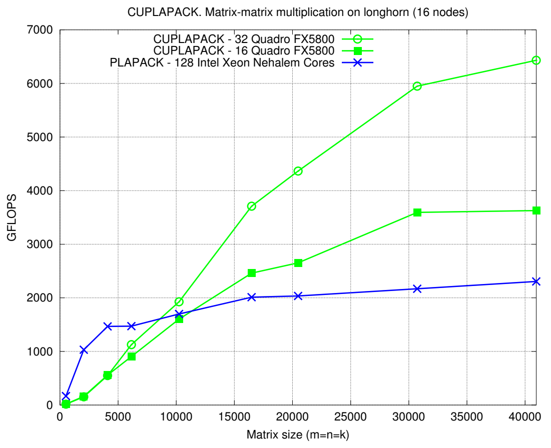
Acknowledgments

The researchers at UJI were supported by the Spanish Ministry of Science and Innovation/FEDER (contract no. TIN2008-06570-C04-01).

We thank NVIDIA for the generous donation of part of the graphics hardware used in the experiments. We also thank the `Texas Advanced Computing Center` (TACC) of The University of Texas at Austin for granting access to the platform where the experiments reported in this paper were conducted.

References

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



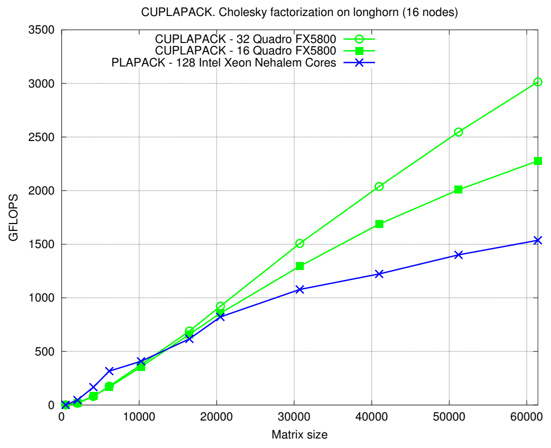


Figure 4: Comparison between PLAPACK and the accelerated version of PLAPACK for the matrix-matrix multiplication (top) and the Cholesky factorization (bottom) on 16 nodes of Longhorn.

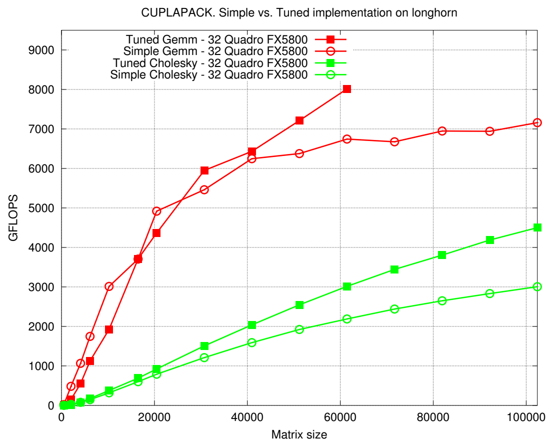


Figure 5: Comparison between the simple and tuned implementation of PLAPACK for the matrix-matrix multiplication and the Cholesky factorization on 16 nodes of Longhorn (using 32 GPUs).

[2] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, and Enrique S. Quintana-Ortí. Solving dense linear systems on graphics processors. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 739–748, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Sergio Barrachina, Maribel Castillo, Francisco D. Igual, Rafael Mayo, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. Exploiting the capabilities of modern GPUs for dense matrix computations. *Concurr. Comput. : Pract. Exper.*, 21(18):2457–2477, 2009.

[4] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker. ScaLAPACK: A scalable linear algebra library for distributed memory concurrent computers. In *Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation*, pages 120–127. IEEE Comput. Soc. Press, 1992.

[5] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton. Petascale computing with accelerators. In *PPoPP '09: The 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 241–150, Raleigh, NC, USA, 2009.

[6] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert van de Geijn. Solving dense linear algebra problems on platforms with multiple hardware accelerators. In *PPoPP '09: The 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 121–129, Raleigh, NC, USA, 2009.

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

[8] Field G. Van Zee. `libflame`*: The Complete Reference*. `www.lulu.com`, 2009.