# Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators

## FLAME Working Note #32

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

**Abstract**

In a previous paper we show how the FLAME methods and tools provide a solution to compute dense dense linear algebra operations on a multi-GPU platform with reasonable performance while requiring little programming effort. In this paper we generalize the approach for systems with multiple hardware accelerators, and incorporate software implementations of standard cache/memory coherence techniques from computer architecture to improve the performance. Our experimental evaluation on an NVIDIA Tesla S870 platform delivers a peak performance well over 400 GFLOPS.

## 1 Introduction

The limitations of current VLSI technology and the desire to transform the ever-increasing number of transistors on a chip dictated by Moore's Law into faster computers has led most hardware manufacturers to design multicore processors and/or specialized hardware accelerators [19]. In response, the computer science community is beginning to embrace (explicit) parallel programming as the means to exploit the potential of the new architectures [1]. While the problem *seems* to be solved from the hardware viewpoint, how to program these new architectures easily and efficiently is the key that will determine their success or failure.

Dense linear algebra has been traditionally used as a pioneering area to conduct research on the performance of new architectures and multicore processors have been no exception. The traditional approach in this problem domain, inherited from the solutions adopted for shared-memory multiprocessors years ago, is based on the use of multithreaded implementations of the BLAS [25, 16, 15]. Code for operations constructed in terms of the BLAS (e.g, for solving a linear system or a linear least-squares problem) extract all the parallelism at the BLAS level. Thus, the intricacies of efficiently utilizing the target architecture are hidden inside the BLAS, and the burden of its parallelization lies in the hands of a few experts with a deep knowledge of the architecture. More recently, the FLAME, PLASMA, and SMPSs projects [11, 12, 13, 29, 30, 31, 9, 8, 4] have advocated for a different approach, extracting the parallelism at a higher level, so that only a sequential tuned implementation of the BLAS is necessary and more parallelism is detected and exploited. Cilk [26] is a precursor of these projects that suffered from not being able to deal with dependencies well. FLAME and PLASMA both focus on dense linear algebra, with the former working at a higher level of abstraction (much in the spirit of object-oriented programming), while the target domain for SMPSs is more general.

Recently, specialized hardware accelerators as the IBM Cell B.E., graphics processors (GPUs), and Field Programmable Gate Arrays (FPGAs) have also attracted the interest of the developers of dense linear algebra libraries [24, 23, 18, 2, 3, 10, 34]. Squeezing these architectures for performance is revealing itself as a task of complexity similar to that of developing a highly tuned implementation of the BLAS for a processor.

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

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

The next evolutionary step has been the construction and use of systems with multiple accelerators: IBM Cell B.E. processors are currently available in the form of blades or PCI-Express accelerator boards, NVIDIA offers nodes with 2 and 4 G80 processors in the Tesla multi-GPU series that can be connected via PCI-Express to a workstation, and ClearSpeed PCI-Express boards are furnished with 2 CSX600 processors. The natural question that arises at this point is how to program these multi-accelerator platforms.

For systems with multiple GPUs, a possibility that has been explored in [34] is to distribute the data among the video memory of the GPUs and code in a style similar to that of the message-passing libraries ScaLAPACK and PLAPACK [14, 33]. We identify two hurdles for this approach:

- While the state-of-the-art numerical methods have not changed, following this approach will require a complete rewrite of dense linear algebra libraries (alike the redesign of LAPACK for parallel distributed-memory architectures that was done in the ScaLAPACK and PLAPACK projects). Therefore, a large programming effort and a considerable amount of funding will be necessary to cover a functionality like that of LAPACK. Note that coding at such low level can be quite complex so that the number of contributors can be significantly reduced.

- The product that is obtained as a result of this style of programming will exhibit a parallelism similar to that of libraries based on multithreaded implementations of the BLAS and far from that demonstrated by the dynamic scheduling techniques in the FLAME, PLASMA, and SMPSs projects. While look-ahead techniques [32] can increase the scalability of this solution to a certain extent, they do so at the cost of a much more complicated coding.

Our approach in this context is fundamentally different. In a previous paper [10], we gave an overview of software tools and methods developed as part of the FLAME project, and we show how, when applied to a platform with multiple GPUs, they provide an out-of-the-box solution that attains reasonable performance almost effortlessly. The key lies in maintaining the independence of code and target architecture, leaving the parallel executing of the operation in the hands of a runtime system [10]. The advantages of this approach are twofold:

- When a new platform appears, it is only the runtime system that needs to be adapted. The routines in the library, which reflect the numerical algorithms, do not need to be modified. For a specific platform, only one runtime is needed to execute all algorithms in parallel.

- The parallelism is extracted by a runtime system which can be easily adapted to exploit multiple accelerators following the ideas that have been shown to be so successful in the multicore arena [11, 12, 13, 29, 30, 31, 9, 8, 4].

While [10] focused in the *programmability* of the solution, this paper makes the following new contributions:

- We give a detailed description of how the programming model accommodates not only the Tesla multi-GPU platforms but generic architectures with multiple hardware accelerators (IBM Cell B.E., NVIDIA GPUs, ClearSpeed boards, etc.).

- We describe several techniques that can be employed to tailor the runtime system for a generic multi-accelerator architecture. Some of these techniques were not employed in [10] and clearly enhance the performance of the code on the multi-GPU platform.

- We use the tuned implementation of the matrix-matrix product described in [34] as well as other implementations of the BLAS kernels for the GPUs, again with a clear impact on the performance of the solution.

- Overall a peak performance of 424 GFLOPS is attained for the (single-precision) Cholesky factorization on a 4-GPU NVIDIA Tesla platform.

The rest of the paper is structured as follows. Section 2 presents the operation considered here: the Cholesky factorization of a dense matrix. Section 3 offers a brief overview of FLAME, the key to easy development of high-performance dense linear algebra libraries that underlies our approach for multi-accelerator platforms. Section 4 describes how the tools in FLAME accommodate for the parallel execution of dense linear algebra codes on these platforms almost effortlessly. More elaborate techniques are presented in Section 5 together with their corresponding performance results. Finally, a few concluding remarks summarize the results in Section 6.

# 2 A Motivating Example

Following [10], in this paper we will consider the Cholesky factorization of an $n \times n$ symmetric positive definite matrix $A$ to illustrate our approach. In this operation, the matrix is decomposed into the product $A = LL^T$, where $L$ is the $n \times n$ lower triangular Cholesky factor. (Alternatively, $A$ can be decomposed as $A = U^T U$, with $U$ being upper triangular.) In traditional algorithms for this factorization, $L$ overwrites the lower triangular part of $A$ while the strictly upper triangular part remains unmodified. Here, we denote this as $A := \{L \backslash A\} = \text{CHOL}(A)$.

# 3 The FLAME Approach to Developing Dense Linear Algebra Libraries

Key elements of FLAME are the high-level notation for expressing algorithms for dense and banded linear algebra operations, the formal derivation methodology to obtain provably correct algorithms, and the high-level application programming interfaces (APIs) to transform the algorithms into codes. FLASH and SuperMatrix are also important components of FLAME that address storage of matrices by blocks and automatic decomposition of linear algebra codes into tasks and dynamic scheduling of these tasks to multithreaded architectures (basically, SMP and multicore processors). We next review all this elements briefly.

## 3.1 FLAME: Formal Linear Algebra Methods Environment

The fundamental innovation that enabled FLAME is the notation for expressing dense and banded linear algebra algorithms. Figure 1 (left) shows a blocked algorithm for computing the Cholesky factorization using the FLAME notation.

The formal derivation methodology consists of a series of steps which, systematically applied, yield families of algorithms (multiple algorithmic variants) for computing an operation [21, 20, 6]. The significance of this for scientific computing is that often different algorithmic variants deliver higher performance on different platforms and/or problem sizes [7, 28]. This derivation of algorithms has also been made mechanical [5].

The FLAME/C API for the C programming language captures the notation in which we express our algorithms. Using this API, the blocked algorithm on the left of Figure 1 can be transformed into the C code on the right of that figure. Note the close resemblance between algorithm and code. As indentation plays an importing role in making the FLAME/C code look like the algorithm, we recommend the use of a high-level mechanical tool like the SPARK webpage (`http://www.cs.utexas.edu/users/flame/Spark/`) which automatically yields a code skeleton.

## 3.2 Storage-by-blocks using FLASH

*Algorithms-by-blocks* [17] view matrices as collections of submatrices and express their computation in terms of these submatrix blocks. Algorithms are then written as before, except with scalar operations replaced by operations on the blocks. Although a number of solutions have been proposed to solve this problem, none of these have yielded a consistent methodology that allows the development of high-performance libraries with functionality that rivals those of LAPACK or FLAME. The problem is primarily one of *programmability*.

**Algorithm:** $A := \text{CHOL\_BLK\_VAR1}(A)$

**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** $b$

**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 $b \times b$

$A_{11} := \{L \backslash A\}_{11} = \text{CHOL\_UNB}(A_{11})$

$A_{21} := L_{21} = A_{21} L_{11}^{-\mathrm{T}}$

$A_{22} := A_{22} - L_{21} L_{12}^{\mathrm{T}} = A_{22} - A_{21} A_{21}^{\mathrm{T}}$

**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**

```
FLA_Error FLA_Chol_blk_var1( FLA_Obj A, int nb_alg )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;
  int b;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    b = min( FLA_Obj_length(ABR), nb_alg );
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                           &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,  b, b, FLA_BR );
    /*-----------------------------------------------*/
    FLA_Chol_unb_var1( A11 );
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              FLA_ONE, A11, A21 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
              FLA_MINUS_ONE, A21, FLA_ONE,       A22 );
    /*-----------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,      FLA_TL );
  }
  return FLA_SUCCESS;
}
```

Figure 1: Blocked algorithm for computing the Cholesky factorization (left) and the corresponding FLAME/C implementation (right).

Our approach to the problem views the matrix as a matrix of smaller matrices using the FLASH API. This view thus yields a matrix hierarchy, potentially with multiple levels. Code for an algorithm-by-blocks for the Cholesky factorization using the FLASH API is given in Figure 2 (left). It may seem that the complexity of the algorithm is merely hidden in the routines `FLASH_Trsm` and `FLASH_Syrk`. The abbreviated implementation of an algorithm-by-blocks for the former is given in Figure 2 (right) while the latter routine has a similar implementation. The reader can see here that many of the details of the FLASH implementation have been buried within the FLASH-aware FLAME object definition.

### 3.3 SuperMatrix runtime system

SuperMatrix extracts the parallelism at a high level of abstraction, decomposing the operation into tasks, identifying the dependencies among these, scheduling them for execution when ready (all operands available/dependencies fulfilled), and mapping tasks to execution units (cores/accelerators) taking into account the target platform. All of this is done without exposing any of the details of the parallelization to the application programmer. The success of this approach has been previously reported in a number of papers [11, 12, 13, 29, 30, 31].

Further details on the operation of SuperMatrix will be illustrated in the next two sections as the strategy to adapt it to a multi-accelerator platform is exposed.

## 4 Adapting FLAME to Platforms with Multiple Accelerators

Previous work on NVIDIA G80 graphics processors and the IBM Cell B.E. view these accelerators as multicore architectures [34, 24] and exploit the parallelism at this level. Our approach is different in that we view one of these accelerators as the equivalent of a single core, for which a tuned "serial" implementation of (specific kernels of the level 3) BLAS is available; our analog of a multicore processor is then a system with multiple accelerators. We therefore exploit parallelism at two levels: at a high level, parallelism due, e.g., to the presence of multiple Cell B.E. or G80 processors is addressed by SuperMatrix. At the low level, the hardware parallelism within the 8 SPUs of a single Cell B.E. or the 128 microcores of a G80 is extracted

```
FLA_Error FLASH_Chol_by_blocks_var1( FLA_Obj A )
{
  FLA_Obj ATL, ATR,      A00, A01, A02,
          ABL, ABR,      A10, A11, A12,
                         A20, A21, A22;

  FLA_Part_2x2( A,    &ATL, &ATR,
                      &ABL, &ABR,    0, 0, FLA_TL );

  while ( FLA_Obj_length( ATL ) < FLA_Obj_length( A ) ) {
    FLA_Repart_2x2_to_3x3(
       ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
    /* ************* */   /* ******************** */
                            &A10, /**/ &A11, &A12,
       ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
       1, 1, FLA_BR );
    /*--------------------------------------------*/
    FLA_Chol_unb_var1( FLASH_MATRIX_AT( A11 ) );
    FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                FLA_ONE, A11,
                         A21 );
    FLASH_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE,
                FLA_MINUS_ONE, A21,
                FLA_ONE,       A22 );
    /*--------------------------------------------*/
    FLA_Cont_with_3x3_to_2x2(
       &ATL, /**/ &ATR,        A00, A01, /**/ A02,
                               A10, A11, /**/ A12,
    /* *************** */   /* ****************** */
       &ABL, /**/ &ABR,        A20, A21, /**/ A22,
       FLA_TL );
  }
  return FLA_SUCCESS;
}
```

```
void FLASH_Trsm_rltn( FLA_Obj alpha, FLA_Obj L,
                                     FLA_Obj B )
/* Special case with mode parameters
     FLASH_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
                 FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                 ...                                 )
   Assumption: L consists of one block and
               B consists of a column of blocks     */
{
  FLA_Obj BT,      B0,
          BB,      B1,
                   B2;

  FLA_Part_2x1( B,    &BT,
                      &BB,    0, FLA_TOP );

  while ( FLA_Obj_length( BT ) < FLA_Obj_length( B ) ) {
    FLA_Repart_2x1_to_3x1( BT,       &B0,
                        /* ** */   /* ** */
                                     &B1,
                           BB,       &B2,    1, FLA_BOTTOM );
    /*--------------------------------------------*/
    FLA_Trsm( FLA_RIGHT, FLA_LOWER_TRIANGULAR,
              FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
              alpha, FLASH_MATRIX_AT( L ),
                     FLASH_MATRIX_AT( B1 ) );
    /*--------------------------------------------*/
    FLA_Cont_with_3x1_to_2x1( &BT,        B0,
                                          B1,
                            /* ** */   /* ** */
                              &BB,        B2,    FLA_TOP );
  }
}
```

Figure 2: FLASH implementation of the Cholesky factorization and the corresponding triangular system solve.

by the BLAS. We hereafter do not pursue further this second level of parallelism and assume the existence of such an implementation of the BLAS.

Our generic multi-accelerator platform consists of a workstation, possibly (but not necessarily) with a multicore CPU, connected to multiple hardware accelerators through a fast interconnect. Processors in the accelerator boards are passive elements that simply wait to be ordered what to do. The workstation RAM (simply RAM from now on) and the memory in the accelerator boards are independent and no hardware memory coherence mechanism is in place (though having one would certainly benefit our approach, as will be reported in the experiments). Communication between the CPU and the accelerators is done via data copies between memories. Communication between two accelerators is only possible through the RAM and is handled by the CPU. This abstract model is general enough to accommodate a workstation connected to a Tesla multi-GPU platform or containing multiple boards with Cell B.E. or ClearSpeed processors.

The SuperMatrix runtime computes the Cholesky factorization by executing the algorithm-by-blocks in Figure 2 (left) in two stages, both executed at run time. During the *analysis stage*, a single thread "executes" the algorithm code, but instead of computing operations immediately as they are encountered, it simply annotates these in a queue of pending tasks. This happens inside the calls to `FLA_Chol_unb_var1`, `FLA_Trsm`, `FLA_Syrk`, and `FLA_Gemm` encountered in the routines `FLASH_Chol_by_blocks_var1`, `FLASH_Trsm`, and `FLASH_Syrk`. As operations are encountered in the code, tasks are enqueued, dependencies are identified, and a DAG (directed acyclic graph) that contains all the dependencies among operations of the overall problem is constructed. To illustrate the outcome of this first stage, the execution of the analysis when the code in Figure 2 is used to factorize the $3 \times 3$ blocked matrix

$$A \rightarrow \begin{pmatrix} \bar{A}_{00} & \bar{A}_{01} & \bar{A}_{02} \\ \bar{A}_{10} & \bar{A}_{11} & \bar{A}_{12} \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} \end{pmatrix}, \tag{1}$$

| Operation/Result | In | | In/out |
|---|---|---|---|
| 1. $\bar{A}_{00} := \text{CHOL}(\bar{A}_{00})$ | | | $\bar{A}_{00}\surd$ |
| 2. $\bar{A}_{10} := \bar{A}_{10}\text{TRIL}(\bar{A}_{00})^{-\text{T}}$ | $\bar{A}_{00}$ | | $\bar{A}_{10}\surd$ |
| 3. $\bar{A}_{20} := \bar{A}_{20}\text{TRIL}(\bar{A}_{00})^{-\text{T}}$ | $\bar{A}_{00}$ | | $\bar{A}_{20}\surd$ |
| 4. $\bar{A}_{11} := \bar{A}_{11} - \bar{A}_{10}\bar{A}_{10}^{\text{T}}$ | $\bar{A}_{10}$ | | $\bar{A}_{11}\surd$ |
| 5. $\bar{A}_{21} := \bar{A}_{21} - \bar{A}_{20}\bar{A}_{10}^{\text{T}}$ | $\bar{A}_{20}$ | $\bar{A}_{10}$ | $\bar{A}_{21}\surd$ |
| 6. $\bar{A}_{22} := \bar{A}_{22} - \bar{A}_{20}\bar{A}_{20}^{\text{T}}$ | $\bar{A}_{20}$ | | $\bar{A}_{22}\surd$ |
| 7. $\bar{A}_{11} := \text{CHOL}(\bar{A}_{11})$ | | | $\bar{A}_{11}$ |
| 8. $\bar{A}_{21} := \bar{A}_{21}\text{TRIL}(\bar{A}_{11})^{-\text{T}}$ | $\bar{A}_{11}$ | | $\bar{A}_{21}$ |
| 9. $\bar{A}_{22} := \bar{A}_{22} - \bar{A}_{21}\bar{A}_{21}^{\text{T}}$ | $\bar{A}_{21}$ | | $\bar{A}_{22}$ |
| 10. $\bar{A}_{22} := \text{CHOL}(\bar{A}_{22})$ | | | $\bar{A}_{22}$ |

Figure 3: An illustration of the DAG resulting from the execution of the SuperMatrix analysis stage for the Cholesky factorization of a $3 \times 3$ matrix of blocks in (1) using the algorithm-by-blocks `FLASH_Chol_by_blocks_var1`. The "$\surd$"-marks denote those operands that are initially available (i.e., those operands that are not dependent upon other operations).

results in the "DAG" implicitly contained in Figure 3.

Once the DAG is constructed, the *dispatch stage* commences. In the SuperMatrix runtime for multithreaded architectures, idle threads monitor the queue of pending tasks till they find a task ready for execution (that is, an operation with all operands available), compute it, and upon completion, update the dependency information in the queue. It is the part of the runtime system responsible for the execution of this second stage that we tailor for multi-accelerator platforms as described next, while the part in charge of the analysis remains unmodified.

Specifically, in our *basic implementation* we run as many threads in the CPU as accelerators are present in the system. When a thread encounters a ready task, it copies the data associated with the operation to the memory of the accelerator, orders it to compute the operation using the appropriate BLAS kernel, and transfers the results back to RAM. We are exposing here a hybrid model of execution where the CPU is responsible for scheduling tasks to the accelerators while tracking dependencies, and the accelerators perform the actual computations. In this hybrid model, tasks that are considered not suitable for execution in the accelerator (due, e.g., to their low complexity or the lack of the appropriate BLAS kernel) can be executed in the CPU. (Hybrid CPU/GPU computation has been previously explored in [2, 3, 10, 34].) Given that the major computational cost is performed by the accelerators in this scheme, the existence of multiple cores in the CPU, though advisable, is not necessary.

This basic implementation incurs an undesirable high amount of data transfers between RAM and the memories of the accelerators so that, unless the cost of communication is negligible, it will surely attain a low practical performance (at this point, we encourage the reader to have a quick glimpse at the line labeled as "Basic implementation" in Figure 4). In the following section we improve the mechanism by including software cache and memory coherence techniques to reduce the number of transfers.

# 5 Improving the Performance

## 5.1 Cache and memory coherence

Standard policies in computer architecture to maintain the coherence between data in the cache of a processor and the main memory are *write-through* (writes to data are immediately propagated to main memory) and *write-back* (data in the main memory is updated only when the cache line where the modified data lie is replaced) [22]. On shared-memory multiprocessors, policies to maintain coherence among the caches of

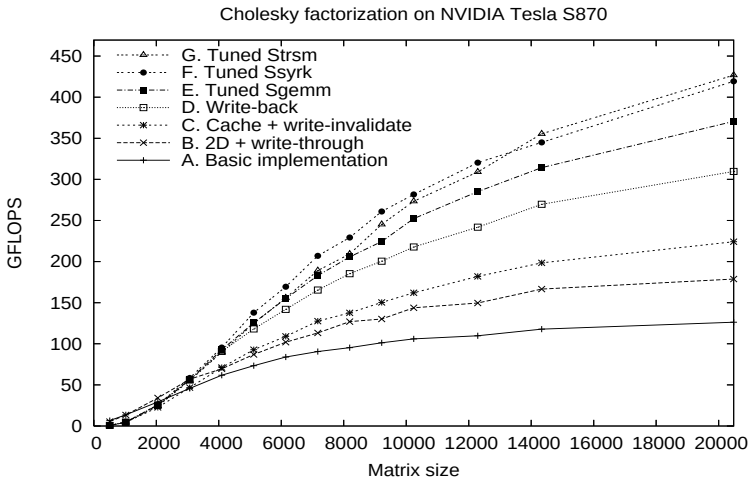


Figure 4: Performance of the blocked algorithm for the Cholesky factorization in the NVIDIA Tesla S870.

the processors are *write-update* (writes to data by one of the processors are immediately propagated to the copies in the caches of the remaining processors) and *write-invalidate* (writes to data by one of the processors invalidate copies of that cache line in the remaining processors) [22].

These policies all aim at reducing the number of data transfers between the cache of the processors and the main memory. Now, at a high level of abstraction, a shared-memory multiprocessor is similar to a workstation connected to multiple accelerators. Each one of the accelerators is the equivalent of one processor with the memory of the accelerator playing the role of the processor cache. The workstation RAM is then the analog of the shared-memory in the multiprocessor. It is not surprising then that we can employ software implementations of standard coherence policies to reduce the number of data transfers between the memory of the accelerators and the RAM of the workstation.

## 5.2 Application to the multi-accelerator platform

The target platform used in the experiments was an NVIDIA Tesla S870 computing system with 4 NVIDIA G80 GPUs and 6 GBytes of DDR3 memory (1.5 GBytes per GPU), which exhibits a theoretical peak performance close to 1400 GFLOPS in single-precision (1 GFLOPS = $10^9$ floating-point arithmetic operations, or flops, per second). The Tesla system is connected to a workstation with one Intel Xeon QuadCore E5405 processor executing at 2.0 GHz with 9 GBytes of DDR2 RAM. The Intel 5400 chipset provides two PCI-Express Gen2 interfaces, for a peak bandwidth of 48 Gbits/second on each interface, to connect with the Tesla. NVIDIA CUBLAS (version 1.1) built on top of the CUDA API (version 1.1) together with NVIDIA driver (171.05) were used in our tests.

When reporting the rate of computation, we consider the cost of the Cholesky factorization to be the standard $n^3/3$ flops for a square matrix of order $n$ so that the GFLOPS rate is computed as $n^3/(3t \times 10^{-9})$, where $t$ equals elapsed time in seconds. However, the actual number of flops in one of the variants that will be evaluated is higher.

Figure 4 reports the performance of the blocked algorithm for the Cholesky factorization in Figure 2 (right) using several variants of the SuperMatrix runtime system and tuned kernels for the G80 BLAS. Unless otherwise stated, these enhancements are incremental so that a variant includes a new strategy plus those of all previous ones. The following seven variants are evaluated:

**A. Basic implementation:** This variant corresponds to the implementation of the runtime system described in Section 4. The Cholesky factorization of the diagonal blocks is computed in the cores of the CPU while all remaining computations (matrix-matrix products, symmetric rank-$k$ updates, and


Figure 5: Cyclic 2-D mapping of the blocks in the lower triangular part of a $4 \times 4$ blocked mapping to the four G80 processors: $G_{00}$, $G_{10}$, $G_{01}$, and $G_{11}$.

triangular system solves) are performed in the G80 processors. FLASH provides transparent storage-by-blocks for the data matrix with one level of hierarchy. The block size is adjusted experimentally.

**B. 2-D + write-through:** In order to improve data locality (and therefore reduce the costly data transfers between the memory of the GPUs), workload is distributed following a cyclic 2-D mapping of the data matrix to a $2 \times 2$ logical grid of the G80s; see Figure 5. (Bidimensional workload distribution in the context of shared-memory multiprocessors has been previously investigated in [27].) In this scheme all operations that compute results which overwrite a given block are mapped to the same G80 processor. Thus, e.g., the updates $\bar{A}_{21} := \bar{A}_{21} - \bar{A}_{20}\bar{A}_{10}^{\mathrm{T}}$ and $\bar{A}_{21} := \bar{A}_{21}\mathrm{TRIL}(\bar{A}_{11})^{-\mathrm{T}}$ are both performed in $G_{01}$. Blocks are thus classified from the viewpoint of a G80 processor into proprietary (owned and written by it) and non-proprietary.

Initially all data blocks reside in the RAM and the memory of the GPUs is empty. When a task is to be computed in a G80 processor, blocks which are not already there are copied to the GPU memory. Proprietary blocks remain in that memory for the rest of the execution of the algorithm while non-proprietary blocks are discarded as soon as the operation is completed. A write-through policy is implemented in software to maintain the coherence between the proprietary blocks in the memory of the GPU and the RAM so that any update of a proprietary block is immediately propagated to the RAM. There is no need to maintain the coherence between the memory of the GPUs and the RAM for non-proprietary blocks as these are read-only blocks. Following the previous example, when the task which computes the update $\bar{A}_{21} := \bar{A}_{21} - \bar{A}_{20}\bar{A}_{10}^{\mathrm{T}}$ is to be computed at $G_{01}$, blocks $\bar{A}_{21}$, $\bar{A}_{20}$, and $\bar{A}_{10}$ are copied to the memory of this GPU; the update is computed and the new contents of $\bar{A}_{21}$ are propagated to RAM. Block $\bar{A}_{21}$ then remains in the GPU memory while the contents of $\bar{A}_{20}$ and $\bar{A}_{10}$ are discarded. Latter, when $\bar{A}_{21} := \bar{A}_{21}\mathrm{TRIL}(\bar{A}_{11})^{-\mathrm{T}}$ is to be computed, only $\bar{A}_{11}$ is copied to the GPU memory as $\bar{A}_{21}$ is already there. Once this second update is computed, following the write-through policy the updated contents of $\bar{A}_{21}$ are sent back to RAM and $\bar{A}_{11}$ is discarded.

Other workload distributions are easily supported by the runtime system and, more important, are transparent to the developer of the algorithms.

**C. Cache + write-invalidate:** The previous strategy reduces the number of transfers between RAM and GPU memory of blocks that are modified, but still produces a large amount of transfers of read-only blocks. In this variant we implement a software cache of read-only blocks in each GPU memory to maintain recently used blocks. With this mechanism in place, e.g., when $G_{10}$ solves the linear systems $\bar{A}_{10} := \bar{A}_{10}\mathrm{TRIL}(\bar{A}_{00})^{-\mathrm{T}}$ and $\bar{A}_{30} := \bar{A}_{30}\mathrm{TRIL}(\bar{A}_{00})^{-\mathrm{T}}$, a copy of $\bar{A}_{00}$ is transferred from RAM to the cache in the GPU memory before the first linear system is solved and remains there for the solution of the second linear system, saving a second transfer.

To complement the cache system, when a task is completed, the thread in the CPU in charge of its execution invalidates all read-only copies of that block in the memory of the "remaining" GPUs (write-invalidate policy).

The replacement policy, currently LRU (least recently used first), and the number of blocks per cache can be easily modified in the runtime system.

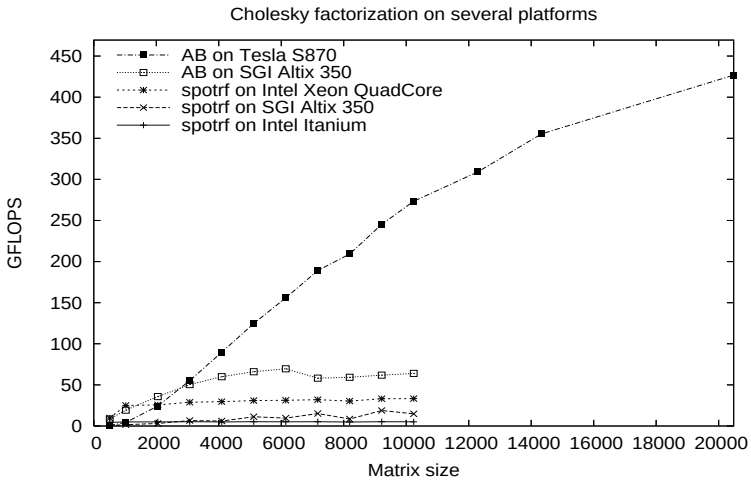


Figure 6: Performance of the Cholesky factorization on several platforms.

**D. Write-back:** The purpose now is to reduce the number of transfers from the memory of the GPUs to RAM that occur when (proprietary) blocks are updated by the G80 processors. For this, write-through is abandoned in favor of a write-back policy which allows inconsistencies between proprietary blocks in the memory of the GPUs and the RAM. Thus, blocks written by a G80 processor are updated in the RAM only when a different G80 (or the GPU) is to compute with them. (Software cache for read-only blocks and the write-invalidate policy are still in place.)

When the execution of the complete algorithm is terminated, the data matrix in the RAM must be updated with the contents of the blocks that have been updated in the memory of the GPU.

**E. Tuned Sgemm:** The matrix-matrix multiplication kernel in CUBLAS 1.1 is replaced by an new implementation to appear in CUBLAS 2.0 and described in [34].

**F. Tuned Ssyrk:** The symmetric rank-$k$ update kernel in CUBLAS 1.1 is replaced by our own implementation based on the tuned matrix-matrix product in CUBLAS 2.0.

**G. Tuned Strsm:** Triangular system solves are computed by first inverting the matrix in the CPU followed by a matrix-matrix multiplication in the GPU. This alternative is proposed in [34] (in the context of the LU factorization with partial pivoting) and incurs an extra useless number of flops that can be balanced by the superior performance of the matrix-matrix multiply kernel. Whether this is an interesting option depends on the performance of the kernels, the cost of communicating the data, and the problem size.

A careful analysis of the numerical stability of this option in the case of the Cholesky factorization is also needed as, in general, solving a triangular linear system in this manner can lead to numerical difficulties.

To put the results into perspective, in Figure 6 we compare the performance with that of optimized implementations of the Cholesky factorization on current high-performance platforms. Single precision was employed in all cases:

- `spotrf` on Intel Itanium: Multithreaded MKL 8.1 implementation of LAPACK routine `spotrf` executed on a 1.5 GHz Intel Itanium2 processor.
- `spotrf` on SGI Altix 350: Multithreaded MKL 8.1 implementation of LAPACK routine `spotrf` executed on a ccNUMA platform with 16 Intel Itanium2 processors at 1.5 GHz which share 32 GBytes of RAM and connected via a SGI NUMAlink.

- `spotrf` on Intel Xeon QuadCore: Multithreaded MKL 10.0 implementation of LAPACK routine `spotrf` executed on a quad-core Xeon SMP workstation that is connected to the Tesla S870.
- `AB` on SGI Altix 350: Our algorithm-by-blocks linked with sequential BLAS in MKL on the the SGI Altix 350.
- `AB` on NVIDIA Tesla S870: Our variant G of the algorithm-by-blocks on the Tesla platform (4 GPUs).

The results show that, on the SGI Altix 350, our algorithm-by-blocks clearly outperforms the highly tuned implementations provided by MKL. On the other hand, the Tesla S870 combined with the algorithm-by-blocks offers a remarkable GFLOPS rate when compared with the multithreaded architectures.

# 6 Conclusions

In this paper we have shown how separation of concerns leads to great flexibility while reducing complexity when porting representative dense linear algebra algorithms to novel architectures. By separating the API for coding algorithms-by-blocks, the part of the runtime system that builds a DAG of operations and tracks the dependencies, and the part of the runtime system that executes operations with blocks, different scheduling heuristics were shown to be easy to implement, allowing customization to what otherwise would have been a very hostile environment: a workstation connected to a multi-GPU accelerator. The particular difficulty of the setting is the fact that the local memory of the GPU is not shared with the host making it necessary to carefully amortize the cost of data transfers.

While the experiments on the paper discuss specifically the multi-GPU NVIDIA Tesla system, the techniques clearly are also applicable to a similar setting where a standard workstation is connected via a fast network to multiple ClearSpeed boards or IBM Cell B.E. accelerators.

Remarkable rates of execution are demonstrated for the important Cholesky factorization operation.

## Additional information

For additional information on FLAME visit `http://www.cs.utexas.edu/users/flame/`.

## Acknowledgements

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714. Additional support came from the *J. Tinsley Oden Faculty Fellowship Research Program* of the Institute for Computational Engineering and Sciences (ICES) at UT-Austin.

The researchers at the Universidad Jaime I were supported by projects CICYT TIN2005-09037-C02-02 and FEDER, and P1B-2007-19 and P1B-2007-32 of the *Fundación Caixa-Castellón/Bancaixa* and UJI.

We thank NVIDIA for the generous donation of equipment that was used in the experiments.

# References

[1] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[2] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Evaluation and tuning of the level 3 CUBLAS for graphics processors. In *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing – PDSEC'08*, 2008. To appear.

[3] S. Barrachina, M. Castillo, F. D. Igual, R. Mayo, and E. S. Quintana-Ortí. Solving dense linear systems on graphics processors. Technical Report ICC 02-02-2008, Universidad Jaume I, Depto. de Ingenieria

y Ciencia de Computadores, February 2008. To appear in Proceedings of the European Conference on Parallel and Distributed Computing – Euro-Par 2008.

[4] Pieter Bellens, Josep M. Pérez, Rosa M. Badía, and Jesús Labarta. CellSs: a programming model for the Cell BE architecture. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing – SC2006*, page 86, New York, NY, USA, 2006. ACM Press.

[5] Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms.* PhD thesis, 2006.

[6] 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.

[7] Paolo Bientinesi, Brian Gunter, and Robert A. Van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Trans. Math. Soft.*, 2009. to appear.

[8] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. LAPACK Working Note 190 UT-CS-07-600, University of Tennessee, September 2007.

[9] Alfredo Buttari, Julien Langou, Jakub Kurzak, , and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. LAPACK Working Note 190 UT-CS-07-598, University of Tennessee, July 2007.

[10] Maribel Castillo, Ernie Chan, Francisco D. Igual, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, Robert van de Geijn, and Field G. Van Zee. Making programming synonymous with programming for linear algebra libraries. FLAME Working Note #31 TR-08-20, The University of Texas at Austin, Department of Computer Sciences, April 2009.

[11] Ernie Chan, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix out-of-order scheduling of matrix operations for SMP and multi-core architectures. In *SPAA '07: Proceedings of the Nineteenth ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 9-11 2007. ACM.

[12] Ernie Chan, Field G. Van Zee, Paolo Bientinesi, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. SuperMatrix: A multithreaded runtime scheduling system for algorithms-by-blocks. In *ACM SIGPLAN 2008 symposium on Principles and Practices of Parallel Programming – PPoPP 2008*, pages 123–132, 2008.

[13] Ernie Chan, Field G. Van Zee, Enrique S. Quintana-Ortí, Gregorio Quintana-Ortí, and Robert van de Geijn. Satisfying your dependencies with SuperMatrix. In *Proceedings of IEEE Cluster Computing 2007*, pages 91–99, September 2007.

[14] 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.

[15] 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.

[16] 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.

[17] Erik Elmroth, Fred Gustavson, Isak Jonsson, and Bo Kagstrom. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[18] Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing – SC2005*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.

[19] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[20] John A. Gunnels. *A Systematic Approach to the Design and Analysis of Parallel Dense Linear Algebra Algorithms*. PhD thesis, Department of Computer Sciences, The University of Texas, December 2001.

[21] 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.

[22] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, 3rd edition, 2003.

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

[24] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving systems of linear equations on the CELL processor using Cholesky factorization. LAPACK Working Note 184 UT-CS-07-596, University of Tennessee, May 2007.

[25] 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.

[26] C. Leiserson and A. Plaat. Programming parallel applications in Cilk. *SINEWS: SIAM News*, 1998.

[27] Bryan A. Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn. Toward scalable matrix multiply on multithreaded architectures. In *European Conference on Parallel and Distributed Computing – Euro-Par 2007*, pages 748–757, February 2007.

[28] Enrique S. Quintana-Ortí and Robert A. van de Geijn. Formal derivation of algorithms: The triangular Sylvester equation. *ACM Trans. Math. Soft.*, 29(2):218–243, June 2003.

[29] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design and scheduling of an algorithm-by-blocks for LU factorization on multithreaded architectures. FLAME Working Note #26 TR-07-50, The University of Texas at Austin, Department of Computer Sciences, September 2007.

[30] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Field G. Van Zee, and Robert A. van de Geijn. Scheduling of QR factorization algorithms on SMP and multi-core architectures. In F. Spies D. El Baz, J. Bourgeois, editor, *16th Euromicro International Conference on Parallel, Distributed and Network-based Processing – PDP 2008*, pages 301–310, 2008.

[31] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Alfredo Remón, and Robert van de Geijn. Supermatrix for the factorization of band matrices. FLAME Working Note #27 TR-07-51, The University of Texas at Austin, Department of Computer Sciences, September 2007.

[32] Peter Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. Technical Report TR-CS-98-07, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, 1998.

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

[34] Vasily Volkov and James Demmel. LU, QR and Cholesky factorizations using vector capabilities of gpus. Technical Report UCB/EECS-2008-XX, EECS Department, University of California, Berkeley, May 2008.