# Solving Dense Linear Systems on Platforms with Multiple Hardware Accelerators

Gregorio Quintana-Ortí Francisco D. Igual
Enrique S. Quintana-Ortí
Departamento de Ingeniería y Ciencia de Computadores
Universidad Jaume I
12.071–Castellón, Spain
{gquintan,figual,quintana}@icc.uji.es

Robert van de Geijn
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
rvdg@cs.utexas.edu

## Abstract

In a previous PPoPP paper we showed how the FLAME methodology, combined with the SuperMatrix runtime system, result in a simple yet powerful solution for programming dense linear algebra operations on multicore platforms. In this paper we provide further evidence that this approach solves the programmability problem for this domain by targeting a more complex architecture, composed of a multicore processor and multiple hardware accelerators (GPUs, Cell B.E., etc.), each with its own local memory, resulting in a platform more reminiscent of a heterogeneous distributed-memory system. In particular, we show that the FLAME programming model accommodates this new situation effortlessly so that no significative change needs to be made to the codebase. All complexity is hidden inside the SuperMatrix runtime scheduling mechanism, which incorporates software implementations of standard cache/memory coherence techniques in computer architecture to improve the performance. Our experimental evaluation on a Intel Xeon 8-core host linked to an NVIDIA Tesla S870 platform with four GPUs delivers peak performances around 550 and 450 (single-precision) GFLOPS for the matrix-matrix product and the Cholesky factorization, respectively, which we believe to be the best performance numbers posted on this new architecture.

***Categories and Subject Descriptors*** D.m [*Software*]: Miscellaneous

***General Terms*** Algorithms, Performance

***Keywords*** GPUs, algorithms-by-blocks, dependency analysis, dynamic scheduling, out-of-order execution

## 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]. How to program these new architectures easily and efficiently is the key that will determine their success or failure. Given that architectures have recently bifurcated and no vendor can even predict what design will dominate five to ten years from now, the design of flexible programming solutions is as important as it ever has been.

Dense linear algebra has been traditionally used as a pioneering area to conduct research on the performance of new architectures and this continues with the recent advent of multicore processors and hardware accelerators like GPUs and Cell B.E. 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 [26, 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, 31, 30, 32, 33, 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 [27] 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.

In the last years, specialized hardware accelerators such as graphics processors (GPUs), Field Programmable Gate Arrays (FPGAs), and the Cell B.E. have also attracted the interest of the developers of dense linear algebra libraries [25, 24, 18, 2, 3, 10, 37]. 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 sequential processor, which typically requires very low-level coding.

The next evolutionary step has been the construction and use of systems with multiple accelerators: NVIDIA offers nodes with multiple Tesla processors (GPUs) in the Tesla series which can be connected via PCI-Express to a workstation and AMD/ATI has recently built a similar system, IBM Cell B.E. processors are currently available in the form of blades or PCI-Express accelerator boards, 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 explored in [37] is to distribute the data among the video memory of the GPUs and code in a message-passing style similar to that of the libraries

ScaLAPACK and PLAPACK [14, 36]. 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 and experts are in short supply.
- The product that is obtained as a result of this style of programming will likely 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 [34] can increase the scalability of this solution to a certain extent, they do so at the cost of a much added complexity.

Our approach in this context is fundamentally different. In previous papers [11, 12, 13, 31, 30, 32, 33], 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 cores/processors, they provide an out-of-the-box solution that attains high performance almost effortlessly. The key lies in maintaining a separation of concern between the code and target architecture by leaving the parallel execution of the operation in the hands of a runtime system. 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.
- The parallelism is orchestrated by a runtime system which can be adapted to exploit different architectures.

While still focused on the *programmability* of the solution, this paper makes the following new contributions:

- We target a fundamentally different architecture, consisting of a multicore processor connected to multiple hardware accelerators, which features properties of an heterogenous distributed-memory multiprocessor. This architecture model is representative of a platform consisting of workstation connected to multiple NVIDIA or AMD/ATI GPUs, IBM Cell B.E. blades, ClearSpeed boards, etc.
- We give a practical demonstration that the FLAME programming model easily accommodates for this generic multi-accelerator architecture, while not requiring a significant modification of the contents of the library.
- We describe how we tailor the runtime system for this generic multi-accelerator architecture by incorporating software implementations of cache/memory coherence techniques from SMP platforms. Altogether, these techniques provide a software distributed-shared memory (DSM) layer, which allows to view the multi-accelerator architecture as a shared-memory multiprocessor. Our experimental results show that the presence of this layer does not decrease performance for large problems.
- We report high performance on an NVIDIA Tesla multi-GPU platform with four G80 processors:
  - A single-precision peak performance of 550 GFLOPS (1 GFLOPS = $10^9$ floating-point arithmetic operations, or flops, per second) is attained for the matrix-matrix product using four G80 processors. Compared with the best implementation of the matrix-matrix product on a single G80 GPU (that of CUBLAS 2.0, based on the implementation in [37]), and measuring the time to transfer the data in both cases, a super-linear speed-up of 5.51 is obtained for the largest problem size.
  - Overall, a (single-precision) peak performance of 460 GFLOPS is attained on the Tesla platform for a more elaborate matrix operation, the Cholesky factorization, with complex data dependencies.

The rest of the paper is structured as follows. Section 2 employs the Cholesky factorization of a dense matrix to offer 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 3 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 4 together with their corresponding performance results. Finally, a few concluding remarks summarize the results in Section 5.

## 2. The FLAME Approach to Developing Dense Linear Algebra Libraries

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 $A := \{L \backslash A\} = \text{CHOL}(A)$.

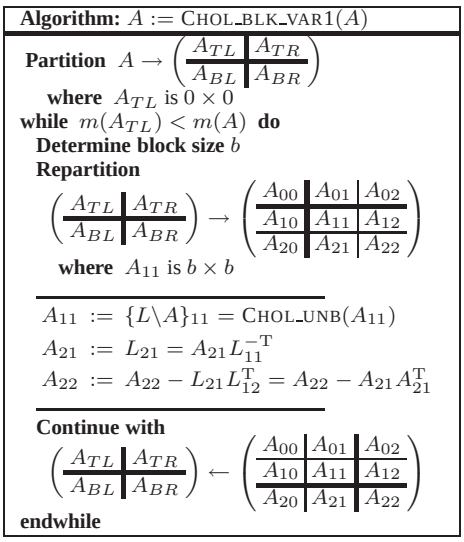
Key elements of FLAME are the high-level notation for expressing algorithms for dense and banded linear algebra operations, the formal derivation methodology to obtain provably correct algorithms, and the high-level application programming interfaces (APIs) to transform the algorithms into codes. FLASH 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). In this section, We briefly review these elements.

### 2.1 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, when 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, 29]. 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.




```
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).

### 2.2 Storage-by-blocks using FLASH

*Algorithms-by-blocks* [17] view matrices as collections of 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 [22, 35, 38], 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*.

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.

### 2.3 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, 31, 30, 32, 33].

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.

## 3. Adapting FLAME to Platforms with Multiple Accelerators

Much work on NVIDIA G80 graphics processors and the IBM Cell B.E. view these accelerators as multicore architectures [37, 25] 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, the presence of multiple accelerators (G80 processors or Cell B.E.) is addressed by SuperMatrix. At the low level, parallelism within the 128 microcores of a G80 or the 8 SPUs of a single Cell B.E. is extracted by the BLAS. We hereafter do not pursue further this second level of parallelism and assume the existence of a tuned 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 explicit 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 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 "symbolically 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

```
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.

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

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

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, 37].) 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.

| 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).

Obviously, 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.

## 4. Improving the Performance

### 4.1 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) [23]. On shared-memory multiprocessors, policies to maintain coherence among the caches of 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) [23].

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.

### 4.2 Application to the multi-accelerator platform

The target platform used in the experiments was an NVIDIA Tesla S870 computing system with four 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. The Tesla system is connected to a workstation with two Intel Xeon QuadCore E5405 processors 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 2.0) built on top of the CUDA API (version 2.0) together with NVIDIA driver (171.05) were used in our tests. MKL 10.0.1 was employed for all computations performed in the Intel Xeon cores. Single precision was employed in all experiments.

When reporting the rate of computation, we consider the cost of the matrix-matrix product and the Cholesky factorization to be the standard $2n^3$ and $n^3/3$ flops, respectively, for square matrices of order $n$. The GFLOPS rate is computed as the number of flops divided by $t \times 10^{-9}$, where $t$ equals the elapsed time in seconds. The cost of all data transfers between RAM and GPU memories is included in the timings.

The top and bottom plots in Figure 4 respectively report the performance of a blocked implementation of the matrix-matrix product and the blocked algorithm for the Cholesky factorization in Figure 2 (right), using several variants of the SuperMatrix runtime system and all four G80 processors of the Tesla. Unless otherwise stated, the enhancements described next are incremental so that a variant includes a new strategy plus those of all previous ones. Although we describe the differences between variants using mostly examples from the Cholesky factorization, the same holds for the matrix-matrix product. Four variants are evaluated in the figure:

**A. Basic implementation:** This variant corresponds to the implementation of the runtime system described in Section 3. In the matrix-matrix product all operations are performed in the G80 processors. For the Cholesky factorization, the diagonal blocks are factorized by the Xeon cores of the CPU while all remaining computations (matrix-matrix products, symmetric rank-$k$ updates, and 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

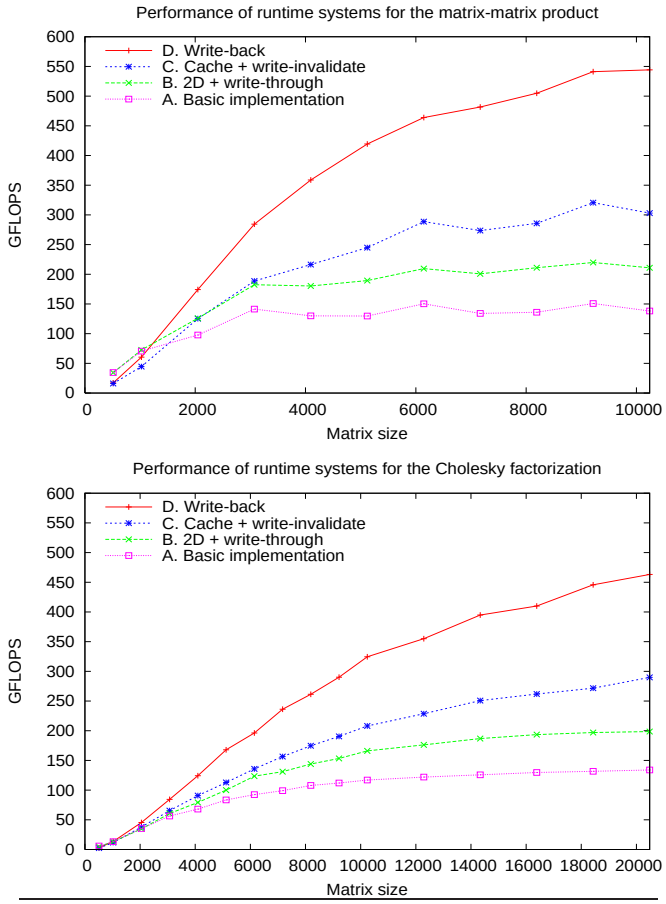



**Figure 4.** Performance of blocked algorithms for the matrix-matrix product (top) and the Cholesky factorization (bottom) using variants A, B, C, and D of the runtime system and the four G80 processors of the Tesla S870.

$$\begin{pmatrix} \bar{A}_{00} & & & \\ \bar{A}_{10} & \bar{A}_{11} & & \\ \bar{A}_{20} & \bar{A}_{21} & \bar{A}_{22} & \\ \bar{A}_{30} & \bar{A}_{31} & \bar{A}_{32} & \bar{A}_{33} \end{pmatrix} \rightarrow \begin{matrix} G_{00} & & & \\ G_{10} & G_{11} & & \\ G_{00} & G_{01} & G_{00} & \\ G_{10} & G_{11} & G_{10} & G_{11} \end{matrix}$$

**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 $G_{11}$.

G80s; see Figure 5. (bidimensional workload distribution in the context of shared-memory multiprocessors has been previously investigated in [28].) In this scheme all operations that compute results which overwrite a given block are mapped to the same G80 processor. Thus, e.g., in the Cholesky factorization 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 = written by it; "owner-computes" rule) 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 for the Cholesky factorization, when the task which performs 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 (block row-wise, block column-wise and cyclic variants) are easily supported by the runtime system and, more important, are transparent to the developer of the algorithms. In our experiments, no major differences were found for the matrix-matrix product and the Cholesky factorization between the performance of the (cyclic) 2-D workload distribution reported in the figure and those of cyclic block row-wise/column-wise layouts.

**C. Cache + write-invalidate:** The previous strategy reduces the number of transfers from RAM to 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 for the Cholesky factorization, 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 which updates a given block 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.

**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.

In summary, the coherence policies and use of cache implemented in variants B, C, and D all aim at reducing the number of data transfers among the different memories present in the system (minimize communication time) while the 2-D distribution pursues a balanced distribution of the work load. A trace of the execution reveals that, from variant A to D, (the number of) block transfers from RAM to the memory of GPUs for the largest problem sizes is reduced from 4096/32760 to 256/2431 for the matrix-matrix product/Cholesky factorization, while the block transfers in the opposite direction are reduced from 12288/11440 to 1280/819. Hereafter all results will refer to variant D of the runtime system.

In Figure 6 we compare the performances of the algorithms-by-blocks for the matrix-matrix product and Cholesky factorization with those of optimized implementations of these operations on current high-performance platforms:

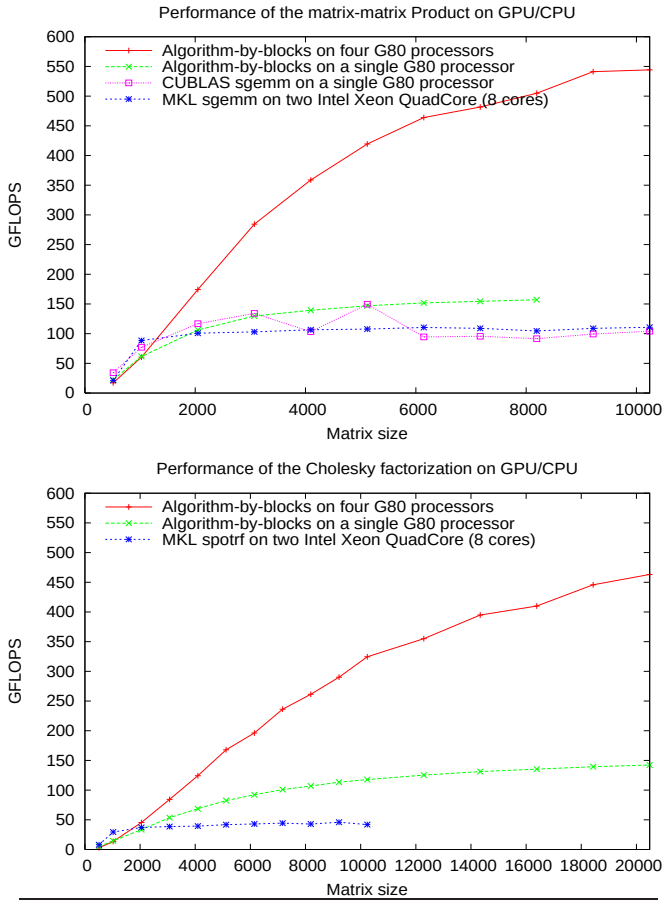



**Figure 6.** Performance of the algorithms-by-blocks for the matrix-matrix product (top) and the Cholesky factorization (bottom) using four G80 processors, the implementation of the matrix-matrix product in CUBLAS 2.0 on one G80 processor, and the implementation of both operations in MKL 10.0.1 using all eight Xeon cores.

- `Algorithms-by-blocks` on four G80 processors: Our algorithms-by-blocks for the two operations, combined with variant D of the runtime system, and executed on the Tesla platform using the four G80 processors.
- `Algorithms-by-blocks` on a single G80 processor: Our algorithms-by-blocks for the matrix-matrix product and Cholesky factorization executed on a single G80 processor of the Tesla platform.
- CUBLAS `sgemm` on a single G80 processor: Implementation of this routine in CUBLAS 2.0 and executed on a single G80 processor. To be consistent with the previous two algorithms, the time to transfer the data from RAM to the memory of the GPUs and retrieve the results is included.
- MKL `sgemm`/`spotrf` on two Intel Xeon QuadCore: Multi-threaded MKL 10.0.1 implementation of the corresponding BLAS/LAPACK routines executed on all eight cores of a workstation with two Xeon Quad-Core processors (details given at the beginning of the section).

The results show that the Tesla S870 combined with the algorithm-by-blocks offers a notable GFLOPS rate when compared with the multicore architecture.

Figures 7 and 8 evaluate the scalability and report the speed-up of the algorithm-by-blocks. No bottlenecks are revealed in the scalability experiment: the performance of the system steadily improves as the number of G80 processors is increased and larger problem sizes report higher execution rates. The speed-ups are calculated comparing the performance attained by the algorithms-by-blocks using 2–4 G80 processors with that of executing same algorithm on a single G80 processor. For the largest problem size of the matrix-matrix product, remarkable speed-ups of 1.83, 2.51, and 3.21 are attained using 2, 3, and 4 G80 processors. Compared with the implementation of the matrix-matrix product in CUBLAS 2.0, and including the time of data transfer between RAM and GPU, the corresponding super-linear speed-ups are 3.14, 4.31, and 5.51. For the largest Cholesky factorization, the results show speed-ups of 1.83, 2.55, and 3.25 using respectively 2, 3, and 4 G80 processors.

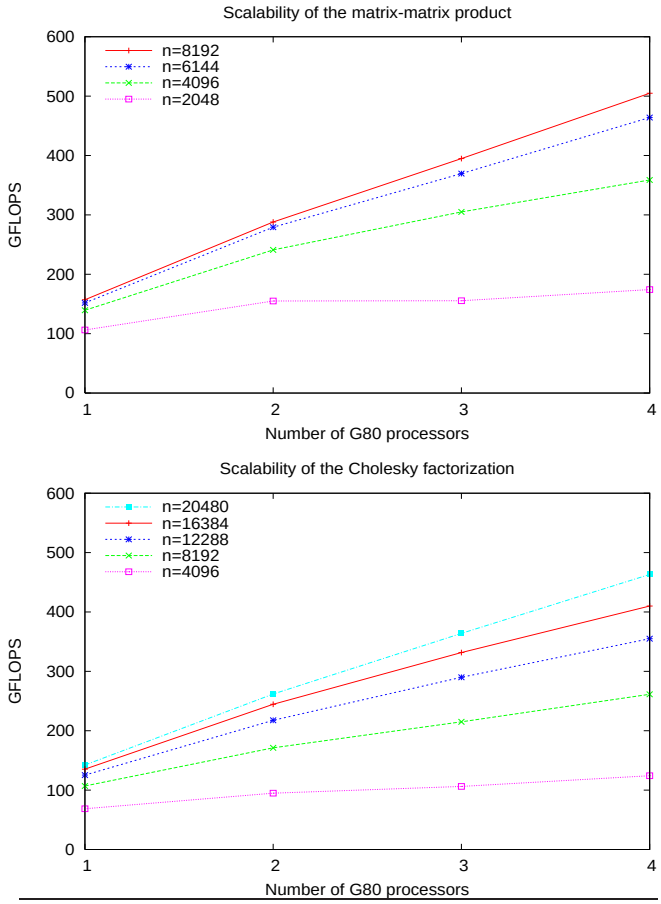



**Figure 7.** Scalability of the algorithms-by-blocks for the matrix-matrix product (top) and the Cholesky factorization (bottom) using 1, 2, 3, and 4 G80 processors.

## 5. 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 architecture-aware 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 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, IBM Cell B.E. accelerators, AMD/ATI GPUs, etc.

Remarkable rates of execution are demonstrated for the matrix-matrix product and the Cholesky factorization operation. Similar results have been obtained for other important BLAS operations as the solution of triangular linear systems and the symmetric rank-$k$ update.

### Additional information

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

## Acknowledgments

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.

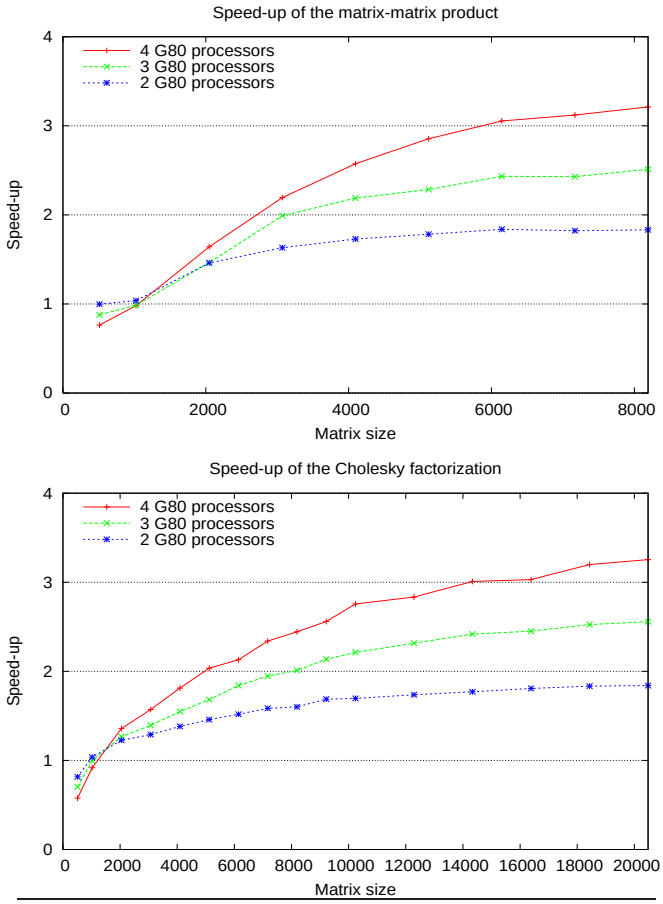



**Figure 8.** Speed-up of the algorithms-by-blocks for the matrix-matrix product (top) and the Cholesky factorization (bottom) using 2, 3, and 4 G80 processors.


[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 Transactions on Mathematical Software*, 35(1):3, July 2008. Article 3, 22 pages.

[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 parallel 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] Jia Guo, Ganesh Bikshandi, Basilio Fraguela, Maria Garzaran, and David Padua. Programming with tiles. In *PPoPP '08: The 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Salt Lake City, UT, USA, 2008.

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

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

[25] 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.

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert van de Geijn, and Field G. Van Zee. Design of scalable dense linear algebra libraries for multithreaded architectures: the LU factorization. In *Workshop on Multithreaded Architectures and Applications – MTAAP 2008*, 2008. CD-ROM.

[32] 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.

[33] 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.

[34] 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.

[35] Vinod Valsalam and Anthony Skjellum. A framework for high-performance matrix multiplication based on hierarchical abstractions, algorithms and optimized low-level kernels. *Concurrency and Computation: Practice and Experience*, 14(10):805–840, 2002.

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

[37] Vasily Volkov and James Demmel. LU, QR and Cholesky factorizations using vector capabilities of GPUs. Technical Report UCB/EECS-2008-49, EECS Department, University of California, Berkeley, May 2008.

[38] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. In *Proceedings of the eighth ACM SIGPLAN symposium on Principles and practices of parallel programming – PPoPP 2001*, pages 24–33, New York, NY, USA, 2001. ACM Press.