SuperMatrix: A Multithreaded Runtime Scheduling System for Algorithms-by-Blocks

FLAME Working Note #25

Ernie Chan*† Field G. Van Zee† Paolo Bientinesi‡ Enrique S. Quintana-Ortí§
Gregorio Quintana-Ortí§ Robert van de Geijn†

Abstract

This paper describes SuperMatrix, a runtime system that parallelizes matrix operations for SMP and/or multi-core architectures. We use this system to demonstrate how code described at a high level of abstraction can achieve high performance on such architectures while completely hiding the parallelism from the library programmer. The key insight entails viewing matrices hierarchically, consisting of blocks that serve as units of data where operations over those blocks are treated as units of computation. The implementation transparently enqueues the required operations, internally tracking dependencies, and then executes the operations utilizing out-of-order execution techniques inspired by superscalar microarchitectures. This separation of concerns allows library developers to implement algorithms without concerning themselves with the parallelization aspect of the problem. Different heuristics for scheduling operations can be implemented in the runtime system independent of the code that enqueues the operations. Results gathered on a 16 CPU ccNUMA Itanium2 server demonstrate excellent performance.

1 Introduction

Architectures capable of simultaneously executing many threads of computation will soon become commonplace. This fact has brought about a renewed interest in studying how to intelligently schedule suboperations to expose maximum parallelism. Specifically, it is possible to schedule some computations earlier, particularly those operations residing along the critical path of execution, to allow more of their dependent operations to execute in parallel. This insight was recognized in the 1960s at the individual instruction level, which led to the adoption of out-of-order execution in many computer microarchitectures [32]. For the dense linear algebra operations on which we will concentrate in this paper, many researchers in the early days of distributed-memory computing recognized that "compute-ahead" techniques could be used to improve parallelism. However, the coding complexity required of such an effort proved too great for these techniques to gain wide acceptance. In fact, compute-ahead optimizations are still absent from linear algebra packages such as ScaLAPACK [12] and PLAPACK [34].

Recently, there has been a flurry of interest in reviving the idea of compute-ahead [1, 25, 31]. Several efforts view the problem as a collection of operations and dependencies that together form a directed acyclic graph (DAG). One of the first to exploit this idea was the Cilk runtime system, which parallelizes divide-and-conquer algorithms rather effectively [26]. Workqueuing [35], when allowed to be nested, achieves a similar effect as Cilk and has been proposed as an extension to OpenMP in form of the `taskq` pragma already supported by Intel compilers. The importance of handling more complex dependencies is recognized in the findings reported by the CellSs project [4], and we briefly discuss this related work in Section 7. Other efforts to apply these techniques to dense matrix computations on the Cell processor [23] are also being pursued [24].

A number of insights gained from the FLAME project allow us to propose, in our opinion, elegant solutions to parallelizing dense and carefully structured sparse linear algebra operations within multithreaded environments. We begin by introducing a notation, illustrated in Figure 1, for expressing linear algebra

*E-mail: echan@cs.utexas.edu
†Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.
‡Department of Computer Science, Duke University, Durham, NC 27708.
§Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I, 12.071–Castellón, Spain.

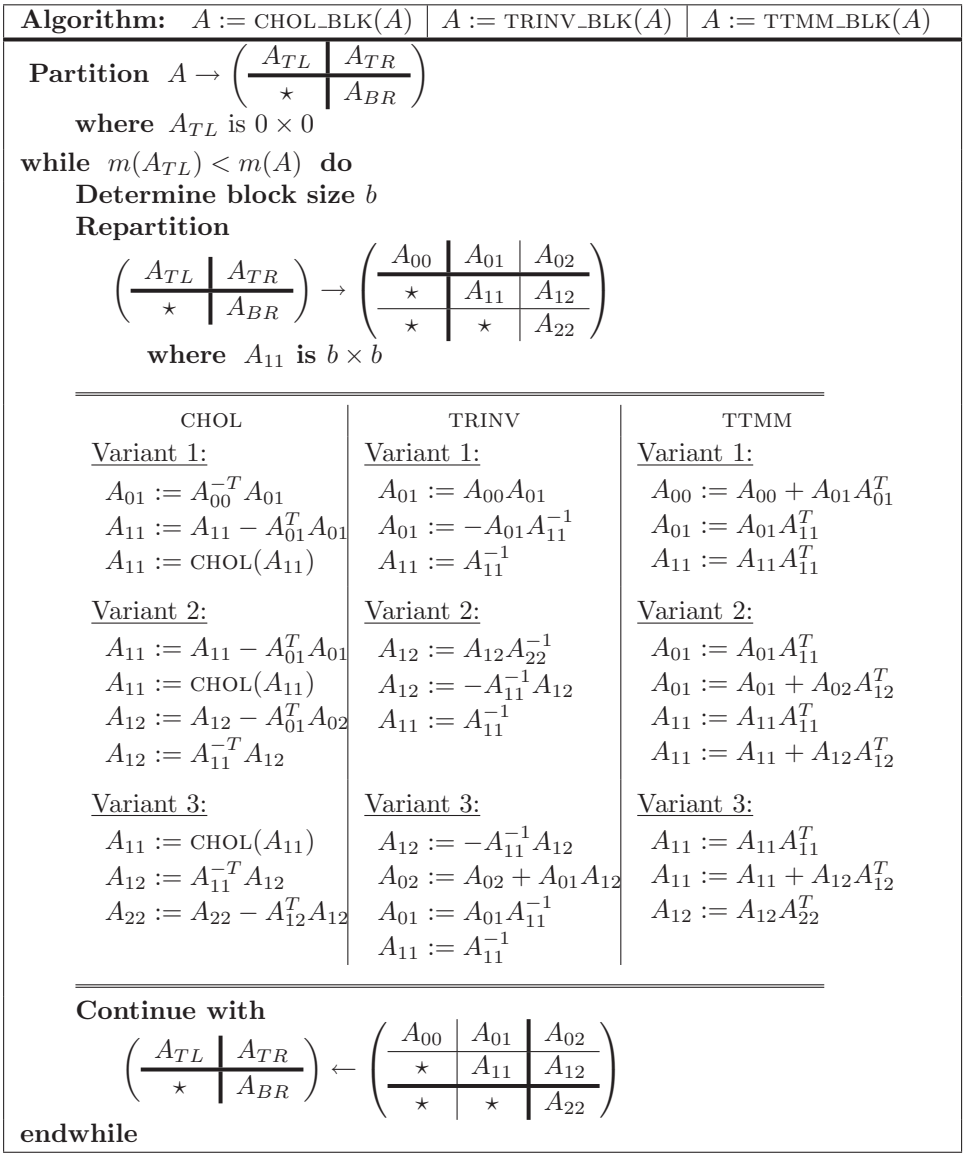


Figure 1: Blocked algorithms for Cholesky factorization, inversion of a triangular matrix, and triangular matrix multiplication by its transpose.

algorithms. This notation closely resembles the diagrams that one would draw to illustrate how the algorithm progresses through the matrices operands [19]. Furthermore, the notation enabled a systematic and subsequently mechanical methodology for generating families of loop-based algorithms given an operation's recursive mathematical definition [5, 6]. Algorithms are implemented using the FLAME/C API [8], which mirrors the notation used to express the algorithms, thereby abstracting away most implementation details such as array indexing. We realized that since the API encapsulates matrix data into typed objects, hypermatrices matrices (matrices of matrices) could easily be represented by allowing elements in a matrix to refer to submatrices rather than only scalar values [11, 13, 14, 21, 29]. This extension to the FLAME/C API known as FLASH [27] greatly reduced the effort required to code algorithms when matrices were stored by blocks, even with multiple levels of hierarchy [22, 33], instead of traditional row-major and column-major orderings.[1] Storing matrices by blocks led to the observation that blocks and computation associated with each block could be viewed as basic units

[1] For a survey of more traditional approaches to expressing and coding recursive algorithms when matrices are stored by blocks, see [16].

of data and computation, respectively [2, 20]. This abstraction allows us to apply superscalar scheduling techniques on operations over blocks via the SuperMatrix runtime system.

The basic idea behind SuperMatrix was introduced in [9], in which we use the Cholesky factorization as a motivating example. Programmer productivity issues and a demonstration that the system covers multiple architectures was the topic of a second paper [10]. For that paper, two of the coauthors implemented in one weekend all cases and all algorithmic variants of an important set of matrix-matrix operations, the level-3 Basic Linear Algebra Subprograms (BLAS) [15], and showed impressive performance on a number of different architectures. This current (third) paper makes the following additional contributions:

- We show in more detail how the FLASH API facilitates careful layering of libraries and how the interface allows SuperMatrix to be invoked transparently by the library developer.
- We discuss the existence of anti-dependencies within linear algebra operations and how SuperMatrix resolves them.
- We illustrate how different algorithmic variants for computing the same operation produce roughly the same schedule with the application of the SuperMatrix runtime system.
- We use a much more complex operation, the inversion of a symmetric positive definite (SPD) matrix, to illustrate these issues.

Together, these contributions further the understanding of the benefits of out-of-order execution through algorithms-by-blocks.

The rest of the paper is organized as follows. In Section 2 we describe inversion of a symmetric positive definite matrix, which we use as a motivating example for the SuperMatrix runtime system in Section 3. We discuss different types of dependencies in Section 4. In Section 5 we show the effect of different algorithmic variants when using SuperMatrix. Section 6 provides performance results. We conclude the paper in Section 7.

2 Inversion of a Symmetric Positive Definite Matrix

Let $A \in \mathbb{R}^{n \times n}$ be an SPD matrix.[2] The traditional approach to implement inversion of an SPD matrix (SPD-INV), e.g., as employed by LAPACK [3], is to compute:

(1) Cholesky factorization $A \rightarrow U^T U$ (CHOL) where U is an upper triangular matrix,

(2) inversion of a triangular matrix $R := U^{-1}$ (TRINV), and

(3) triangular matrix multiplication by its transpose $A^{-1} := RR^T$ (TTMM).

If only the upper triangular part of A is originally stored, each stage can overwrite that upper triangular part of the matrix. In the remainder of this section we briefly discuss these three operations (sweeps) separately.

For a more thorough discussion, we refer to [7] where the inversion of an SPD matrix is used to illustrate that the ideal choice of algorithm for a given operation is greatly affected by the characteristics of the target platform. The present paper uses the same operation to illustrate issues related to algorithms-by-blocks and out-of-order execution. We compare and contrast these two papers in the conclusion.

It is well understood that in order to attain high performance, matrix algorithms of this kind must be cast in terms of blocked computations so that the bulk of the computation is in matrix-matrix multiplication (level-3 BLAS). In Figure 1 we give three blocked algorithmic variants for each of the three sweeps. The algorithms use what have become standard FLAME notation. The thick and thin lines have semantic meaning and capture how the algorithms move through the matrices, exposing submatrices on which computation occurs. Each algorithm overwrites the original matrix A. All three algorithms are captured concisely in one figure, easily allowing us to compare and contrast. Thus, to understand what computation is performed in the body of the loop by, e.g., blocked Variant 3 for Cholesky factorization, one looks under the column marked CHOL in the row marked Variant 3, ignoring all other operations.

A few comments are in order. In many of the operations it is implicitly assumed that a matrix is upper triangular and/or only the upper triangular part of a matrix is updated. For any operation of the form

[2] For the purpose of this discussion, A being SPD means that all the Cholesky factorization algorithms execute to completion, generating a unique nonsingular factor U.

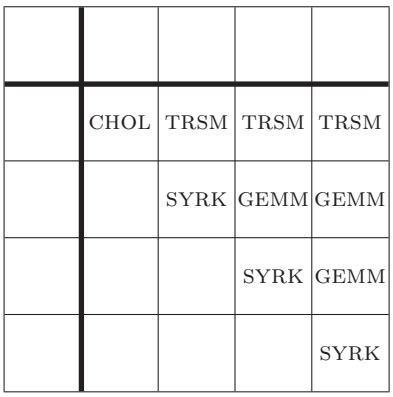


Figure 2: The tasks performed in the second iteration of Cholesky factorization Variant 3 on a 5×5 matrix of blocks.

$Y := B^{-T}Y$ it is implicitly assumed that B is upper triangular and that Y is updated by the solution of $B^T X = Y$, also known as a triangular solve with multiple right-hand sides. A similar comment holds for $Y := YB^{-1}$.

In [7] we show how appropriately chosen variants for each of these three sweeps can be combined into a single sweep algorithm by rearranging the order of computations. This rearrangement yields better load balance on distributed-memory architectures. Since the SuperMatrix system rearranges operations automatically, there is no need to discuss that in the current paper.

3 Algorithms-By-Blocks

The algorithms in Figure 1 move through the matrix, exposing a new $b \times b$ submatrix A_{11} during each iteration. If the matrix is viewed as a matrix of $b \times b$ submatices, each stored as blocks, then the algorithms move through the matrix by exposing whole blocks. In general, all operations that are performed can be viewed as sub-operations (tasks) on blocks. For example in the Cholesky factorization Variant 3:

- $A_{11} := \text{CHOL}(A_{11})$ requires a Cholesky factorization of the block A_{11}.
- $A_{12} := A_{11}^{-T} A_{12}$ requires independent triangular solves with multiple right-hand sides (TRSM) with A_{11} and the blocks that comprise A_{12}.
- $A_{22} := A_{22} - A_{12}^T A_{12}$ requires each of the blocks on or above the diagonal of A_{22} to be updated by a matrix-matrix multiplication (GEMM) for the off-diagonal blocks and a symmetric rank-k update (SYRK) for the diagonal blocks.

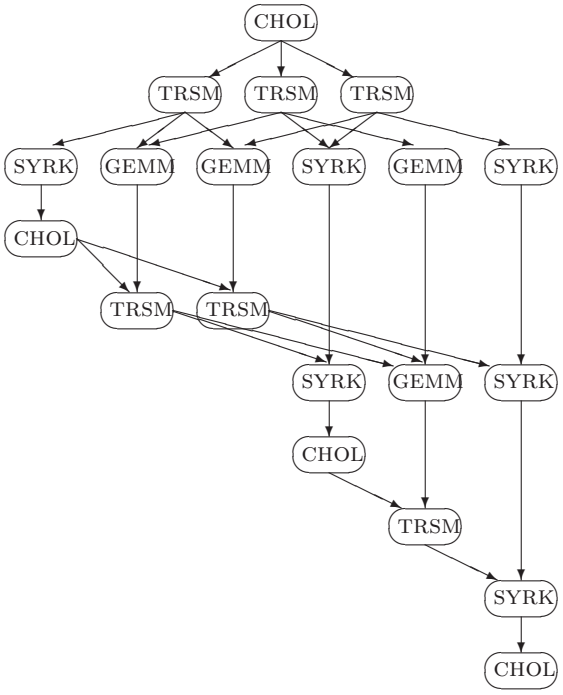


Figure 3: The directed acyclic graph formed by the dependencies of Cholesky factorization Variant 3 on a 4×4 matrix of blocks.

We visualize these tasks performed in the second iteration of Cholesky factorization Variant 3 on a 5×5 matrix of blocks in Figure 2. Next, we illustrate the directed acyclic graph formed by the dependencies of Cholesky factorization Variant 3 on a 4×4 matrix of blocks in Figure 3. Note that the tasks depicted in Figure 2 correspond to the tasks in the first three levels of the DAG in Figure 3.

In Figure 4 we show how the FLASH API for computing over matrices stored by blocks allows the details of the implementation to be hidden. A few comments are due:

- The API is designed and the code is typeset so that the implementation closely mirrors the algorithms in Figure 1.
- The calls `FLASH_Chol`, `FLASH_Trinv`, and `FLASH_Ttmm` each enqueue a single task and internally register their dependencies.
- The routines `FLASH_Trsm`, `FLASH_Syrk`, `FLASH_Gemm`, and `FLASH_Trmm` are coded in the same style of Figure 4. These routines decompose themselves into their component tasks which are all enqueued.

The details of enqueueing and the specification of dependencies are determined when the aforementioned

```
FLA_Error FLASH_SPD_inv_u_op( int op, 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 );

      /*-----------------------------------------------*/
      switch ( op )
      {
      FLASH_Chol_op:   /* Variant 3 */
         FLASH_Chol( FLA_UPPER_TRIANGULAR, A11 );
         FLASH_Trsm( FLA_LEFT, FLA_UPPER_TRIANGULAR,
                     FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                     FLA_ONE, A11, A12 );
         FLASH_Syrk( FLA_UPPER_TRIANGULAR, FLA_TRANSPOSE,
                     FLA_MINUS_ONE, A12, FLA_ONE, A22 );
         break;
      FLASH_Trinv_op:  /* Variant 3 */
         FLASH_Trsm( FLA_LEFT, FLA_UPPER_TRIANGULAR,
                     FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                     FLA_MINUS_ONE, A11, A12 );
         FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                     FLA_ONE, A01, A12, FLA_ONE, A02 );
         FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                     FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                     FLA_ONE, A11, A01 );
         FLASH_Trinv( FLA_UPPER_TRIANGULAR,
                      FLA_NONUNIT_DIAG, A11 );
         break;
      FLASH_Ttmm_op:   /* Variant 1 */
         FLASH_Syrk( FLA_UPPER_TRIANGULAR, FLA_NO_TRANSPOSE,
                     FLA_ONE, A01, FLA_ONE, A00 );
         FLASH_Trmm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                     FLA_TRANSPOSE, FLA_NONUNIT_DIAG,
                     FLA_ONE, A11, A01 );
         FLASH_Ttmm( FLA_UPPER_TRIANGULAR, A11 );
         break;
      }
      /*-----------------------------------------------*/

      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 4: SuperMatrix implementation of selected variants for all three operations required for inversion of a symmetric positive definite matrix.

`FLASH_*` routines are called.

The inversion of an SPD matrix may now be performed with:

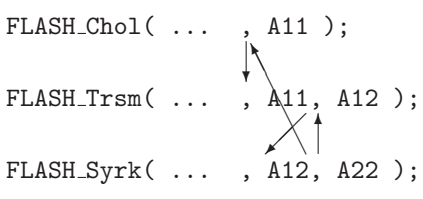
```
FLASH_SPD_inv_u_op( FLASH_Chol_op, A );
FLASH_SPD_inv_u_op( FLASH_Trinv_op, A );
FLASH_SPD_inv_u_op( FLASH_Ttmm_op, A );
FLASH_Queue_exec( );
```

Stage	Scheduled Tasks			
1	CHOL			
2	TRSM	TRSM	TRSM	TRSM
3	SYRK	GEMM	SYRK	GEMM
4	GEMM	SYRK	GEMM	GEMM
5	GEMM	SYRK	CHOL	TRSM
6	TRSM	TRSM	TRSM	TRSM
7	TRSM	TRSM	TRINV	SYRK
8	GEMM	SYRK	GEMM	GEMM
9	SYRK	TTMM	CHOL	TRSM
10	TRSM	TRSM	TRSM	TRSM
11	GEMM	GEMM	GEMM	SYRK
12	GEMM	SYRK	TRSM	CHOL
13	TRSM	TRSM	TRINV	SYRK
14	TRSM	GEMM	GEMM	GEMM
15	GEMM	TRMM	SYRK	TRSM
16	TRSM	TTMM	CHOL	TRSM
17	SYRK	TRINV	GEMM	SYRK
18	GEMM	GEMM	GEMM	TRMM
19	TRMM	TRSM	TRSM	TRSM
20	TRSM	TRSM	TRSM	TRSM
21	TTMM	SYRK	GEMM	SYRK
22	TRINV	GEMM	GEMM	TRINV
23	SYRK	SYRK	GEMM	SYRK
24	TRMM	GEMM	TRMM	GEMM
25	TRMM	SYRK	GEMM	GEMM
26	TTMM	GEMM	TRMM	TRMM
27	SYRK	TRMM		
28	TRMM			
29	TTMM			

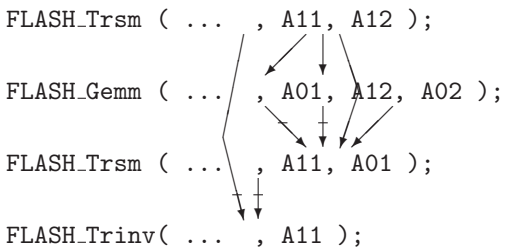
Figure 5: Simulated SuperMatrix execution of inversion of a symmetric positive definite matrix on a 5×5 matrix of blocks using four threads. Each column represents the tasks executed on separate threads.

Once all tasks are enqueued, `FLASH_Queue_exec` triggers the dependency analysis, which then allows the SuperMatrix runtime system to execute tasks out-of-order.

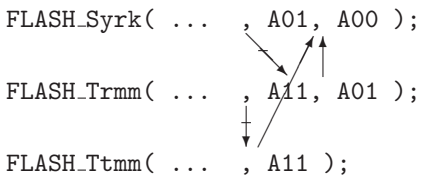
In Figure 5, we show the simulated SuperMatrix execution of SPD-INV on a 5×5 matrix of blocks using four threads. Each column represents tasks that execute on separate threads. For illustration purposes, this simulation assumes that each thread performs lock-step synchronization after executing a single task, which we denote as a single stage. Even with this restrictive simulation, SuperMatrix achieves near-perfect load balance among threads. Such high efficiency is possible because the SuperMatrix runtime system exposes concurrency between the three component operations,



(a) Cholesky factorization



(b) Inversion of a triangular matrix



(c) Triangular matrix multiplication by its transpose

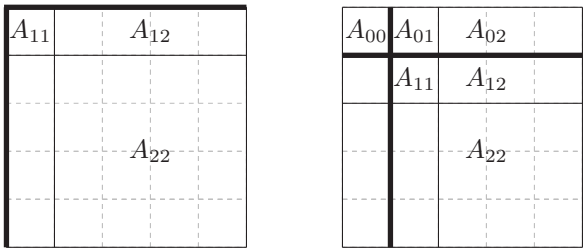
Figure 6: Dependencies within the `while` loop of each operation shown in Figure 4. The plain arrows denote flow dependencies, and the arrows with a dash denote anti-dependencies.

which is unattainable by only parallelizing the three sweeps separately.

Figure 5 depicts a prime example of out-of-order scheduling. Stage 21 contains a TTMM task corresponding to a sub-operation normally found in the third sweep. This TTMM task is scheduled to execute *before* a TRINV task in Stage 22 that originated from the second sweep.

4 Flow vs. Anti-Dependencies

With respect to its application of superscalar execution techniques, SuperMatrix raises the level of abstraction by treating sub-operations over submatrix blocks as the fundamental unit of computation. Similar in concept to instruction-level parallelism, dependencies between tasks that read from and write to blocks determine the potential for concurrency. When parallelizing linear algebra operations, it is natural to focus on flow dependencies and ignore anti-dependencies.



Iteration 1 Iteration 2

Figure 7: First two iterations of a FLAME algorithm on a 5×5 matrix of blocks.

Even though the SuperMatrix runtime system automatically detects all three types of dependencies, we take special care to detect these dependencies in algorithms coded using the FLAME/C API for the illustrative elements of this section.

In all the level-3 BLAS operations, the last operand is the only matrix that is overwritten by the operation whereas all preceeding operands are strictly inputs. LAPACK functions also follow this convention. We leverage this feature of the interface when detecting dependencies.

Figure 6 shows flow and anti-dependencies within the three sweeps of SPD-INV. The plain arrows denote flow dependencies, and the arrows with a dash denote anti-dependencies. Since all sub-operations in Figure 6 lie within the `while` loop of Figure 4, we illustrate both intra- and inter-iterational dependencies.

4.1 Flow dependencies

Flow dependencies, often referred to as "true" dependencies, create situations where a block is read by one task after it is written by a previous task (read-after-write).

```
S1:  A = B + C;
S2:  D = A + E;
```

In this simple example, statement `S1` must complete execution and output the value of `A` before `S2` can begin.

Intra-iterational flow dependencies are easily detectable in operations implemented using the FLAME/C API since the same submatrix view is used in multiple code locations. In Figure 6(a), `A11` is first written by `FLASH_Chol` and then read by `FLASH_Trsm`.

Inter-iterational flow dependencies can be more difficult to recognize systematically because different submatrix views may reference the same block across multiple iterations. We show the blocks comprising different views in the first two iterations of a typical FLAME

algorithm in Figure 7. The top left block of `A22` becomes `A11` in the next iteration, which leads us to the flow dependency between `FLASH_Syrk` and `FLASH_Chol` within the Cholesky factorization shown in Figure 6(a).

Though not shown, flow dependencies form self loops on `FLASH_Syrk` in Figure 6(a) and Figure 6(c). That is to say, in the case of Cholesky factorization, the `FLASH_Syrk` from iteration i must complete before the `FLASH_Syrk` from iteration $i + 1$ may begin.

The DAG in Figure 3 corresponds to the dependencies shown in Figure 6(a) where all the sub-operations are decomposed into their component tasks when executed on a 4×4 matrix of blocks. We can view the DAG as the dynamic loop rolling of tasks specified by the implementation of Cholesky factorization in Figure 4.

Detecting dependencies within FLAME/C algorithms can be laborious, as it requires complete knowledge of matrix partitioning to determine exactly which regions of the matrix are being referenced. However, the FLASH extension to FLAME/C provides a distinct advantage by clearly delimiting the blocks referenced by each submatrix view. Identifying dependencies within algorithms coded using traditional indexing would likely be difficult and error-prone, at best.

4.2 Anti-dependencies

Anti-dependencies occur when a task must read a block before another task can write to the same block (write-after-read).

```
S3:  F = A + G;
S4:  A = H + I;
```

Here, `S3` must read the value of `A` before `S4` overwrites it.

In the operations examined in [9, 10], we only encountered flow dependencies. However, TRINV and TTMM exhibit both flow and anti-dependencies. Both types of dependencies must be obeyed in order to correctly parallelize SPD-INV.

As with flow dependencies, intra-iterational anti-dependencies are straightforward to locate. `A01` is read by `FLASH_Syrk` but then overwritten by `FLASH_Trmm` in Figure 6(c).

However, inter-iterational anti-dependencies can be quite difficult to identify. In Figure 7, we can see that the leftmost block of `A12` becomes `A01` in the next iteration. This example results in an anti-dependency between `FLASH_Gemm` and `FLASH_Trsm` in Figure 6(b).

The SuperMatrix runtime system detects anti-dependencies while tasks are being enqueued. For each block, the runtime system maintains a temporary queue that tracks which tasks must read the block's data. When the system detects that a task writes to the block in question, all previously inspected tasks on the temporary queue are marked with anti-dependencies. The block's temporary queue is cleared after the anti-dependencies are recorded.

4.3 Output dependencies

Output dependencies, the final class of dependencies, create situations where the result of one operation is overwritten by the result of subsequent computation (write-after-write).

```
S5: A = J + K;
S6: A = L + M;
```

In this example, `S6` must be executed after `S5` to ensure that `A` contains the correct value.

No BLAS operation strictly overwrites a matrix where writes to the matrix are performed without first reading its contents since the last operand is always an input/output operand. The only exception is the copy operation implemented in FLAME via `FLA_Copy`.

Even though copying rarely occurs in linear algebra operations, the QR factorization is one such example where a temporary matrix is used, thereby creating instances of output dependencies. We have adapted the SuperMatrix mechanism to compute the QR factorization [30], but that topic is beyond the scope of this paper.

5 Scheduling Different Algorithmic Variants

When linked to a serial BLAS implementation, different algorithmic variants for computing a particular linear algebra operation often yield significantly different performance results. The better performing algorithms make efficient use of the architecture's memory hierarchy. When linking the same algorithms to multithreaded BLAS, the difference in performance is even more pronounced.

5.1 Mulithreaded BLAS

We show the different problem instances of symmetric rank-k update encountered in Cholesky factorization Variant 1 and Variant 3 in Figure 8(a) and (b), respectively, on a 4×4 matrix of blocks. The computation shown in Figure 8(a) takes the form of a generalized blocked dot product, which does not parallelize well. After decomposing this problem instance into its

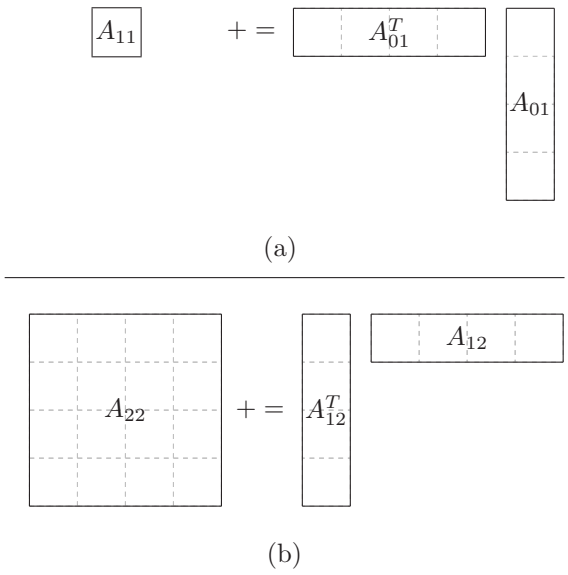


Figure 8: Problem instances of symmetric rank-k update encountered by Cholesky factorization Variant 1 in (a) and Variant 3 in (b) on a 4×4 matrix of blocks.

component tasks, flow dependencies exist between every task pair because each blocked sub-operation overwrites the single output matrix block. By contrast, the rank-k update in Figure 8(b) can be parallelized quite easily. Here, each component task overwrites a separate block of the output matrix, allowing each task to execute independently. The bulk of the computation in Cholesky factorization occurs in the symmetric rank-k update, so implementations of Variant 3 usually attain much higher performance than those of Variant 1.

Identifying algorithmic variants with subproblems that parallelize well requires careful analysis. A poor choice of algorithm can result in near-serial performance even when linking to the best multithreaded BLAS libraries.

5.2 SuperMatrix

SuperMatrix essentially performs a dynamic loop unrolling of all tasks executing on individual submatrix blocks. We can view different algorithmic variants simply as permutations of subproblems. For example, Cholesky factorization Variant 1 performs TRSM followed by SYRK and then CHOL whereas Variant 3 rearranges the order such that CHOL occurs at the top of the loop. The problem instances of SYRK shown in Figure 8 no longer exist in isolation but are decomposed into their component tasks with dependencies between tasks from other subproblems.

The differences between variants are limited to the order in which subproblems are called and the submatrix views that are referenced. The tasks executed on each submatrix block remain the same since the dependencies remain invariant across all algorithmic variants. As a result SuperMatrix produces nearly identical schedules regardless of the algorithmic variant being used to implement an operation.

Performance may vary slightly among different algorithmic variants implemented using SuperMatrix. Tasks are enqueued onto a global task queue where the dependencies that form the DAG are embedded into the task structures. Even though different variants form similar DAGs, the order in which tasks appear in the task queue will vary drastically. As a result, the order in which tasks are executed and therefore the order in which their operands' submatrix blocks are referenced will also vary. Several secondary effects such as the level-2 cache performance may also affect the overall performance of different algorithmic variants.

We intend to explore this topic more formally in a future paper with the context of scheduling tasks in a DAG.

6 Performance

In this section, we show that SuperMatrix yields a performance improvement for each of the three sweeps. We observe further performance improvements when the three sweeps are combined and tasks are scheduled out-of-order.

6.1 Target architecture

All experiments were performed on an SGI Altix 350 server using double-precision floating-point arithmetic. This ccNUMA architecture consists of eight nodes, each with two 1.5 GHz Intel Itanium2 processors, providing a total of 16 CPUs and a peak performance of 96 GFLOPs/sec. (96×10^9 floating point operations per second). The nodes are connected via an SGI NUMAlink connection ring and collectively provide 32 GB (32×2^{30} bytes) of general-purpose physical RAM. The OpenMP implementation provided by the Intel C Compiler served as the underlying threading mechanism used by SuperMatrix. Performance was measured by linking to two different high-performance implementations of the BLAS: the GotoBLAS 1.15 [17] and Intel MKL 8.1 libraries.

6.2 Implementations

We report the performance (in GFLOPs/sec.) of three different implementations for CHOL, TRINV, TTMM, and

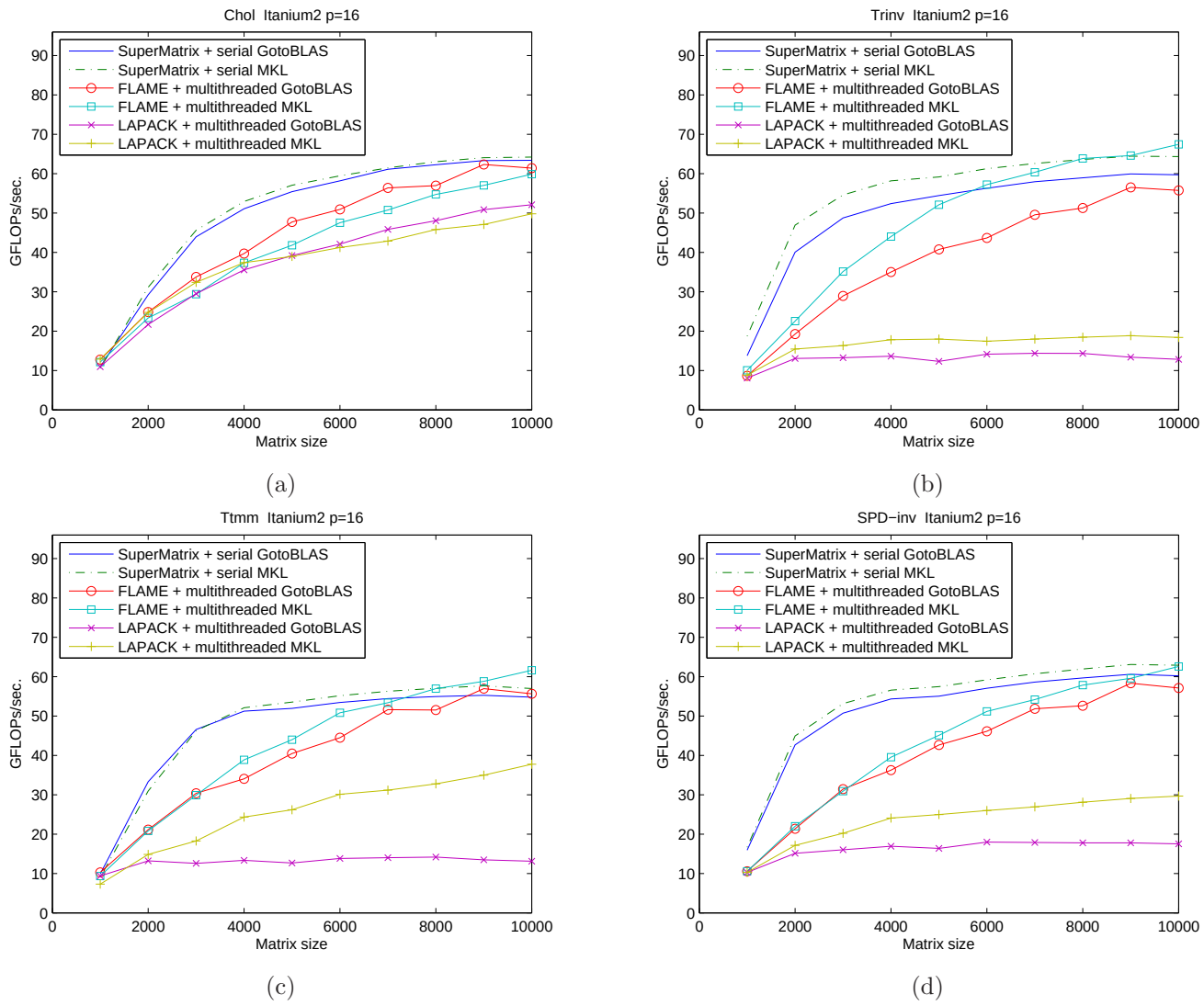


Figure 9: Performance on the Itanium2 machine using 16 CPUs linked with the GotoBLAS 1.15 and MKL 8.1 libraries.

SPD-INV. An algorithmic block size of 192 was used for all experiments.

- **SuperMatrix + serial BLAS**
  The SuperMatrix implementation of each operation is given in Figure 4. For the execution of individual tasks on each thread, we linked to serial BLAS libraries.

  The results reflect our simplest SuperMatrix mechanism, which is explained in detail in [10]. We did not use advanced scheduling techniques such as *data affinity* described in [9].

- **FLAME + multithreaded BLAS**
  The sequential implementations of `FLA_Chol`, `FLA_Trinv`, and `FLA_Ttmm` provided by libFLAME 1.0 are linked with multithreaded BLAS libraries.

  libFLAME implements Variant 3 for CHOL, Variant 3 for TRINV and Variant 1 for TTMM, which are the same algorithms implemented with SuperMatrix.

- **LAPACK + multithreaded BLAS**
  We linked the sequential implementations of `dpotrf`, `dtrtri`, `dlaaum` provided by LAPACK 3.0 with multithreaded BLAS libraries. We modified the routines to use an algorithmic block size of 192 instead of the block sizes provided by LAPACK's `ilaenv` routine.

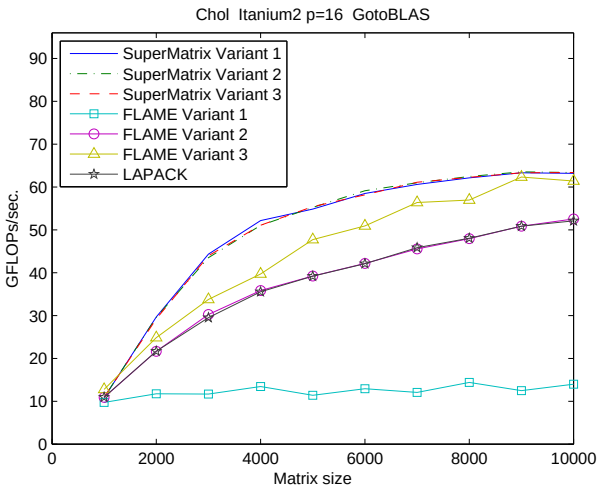


Figure 10: Performance of all three variants for Cholesky factorization on the Itanium2 machine using 16 CPUs linked with the GotoBLAS 1.15 library.

LAPACK implements Variant 2 for CHOL, Variant 1 for TRINV and Variant 2 for TTMM.

We performed experiments computing with both the upper and lower triangular parts of an SPD matrix but only included the results for upper triangular. The upper triangular results provided the best performance from the LAPACK implementations whereas lower triangular results were reported in [9, 10].

6.3 Results

Performance results when linking to the GotoBLAS and MKL libraries are reported in Figure 9. Several comments are in order:

- In this multithreaded setting, LAPACK implements sub-optimal algorithms for each of the three sweeps. By simply choosing a different algorithmic variant, the corresponding experiments combining FLAME with multithreaded BLAS provide greater performance for each sweep.

- Since each implementation used a block size of 192, experiments run with GotoBLAS and MKL generally yield similar performance curves with the exception of TTMM in LAPACK, shown in Figure 9(c).

  In [9] we tuned the block size for each problem size and found that serial MKL outperformed GotoBLAS on small matrices, which are used heavily in SuperMatrix, while multithreaded GotoBLAS provided higher performance for large matrices.

  In [10] we reported performance results across several different computer architectures linked with various vendors' BLAS libraries. Notably, those results demonstrated that GotoBLAS typically provides the best performing serial and multithreaded BLAS implementation. The one exception to this observation is Intel's MKL, which narrowly but consistently outperforms GotoBLAS on Itanium2 architectures.

- In order to attain high performance, BLAS libraries pack matrices into contiguous buffers to obtain stride one access during execution [18]. For large matrices, this packing cost is amortized over enough computation to minimize the overall impact on performance. This cost is more visible when performing tasks over blocks, as occurs in SuperMatrix. Making matters worse, many blocks are accessed by several tasks, requiring those blocks to be repeatedly packed and unpacked.

  In [28] the authors show how to avoid these redundant packing operations in the context of general matrix-matrix multiplication. We suspect these techniques can be extended to SuperMatrix.

- SuperMatrix performance ramps up much faster for all operations, especially for SPD-INV in Figure 9(d). The asymptotic performance of the best sequential implementations linked with multithreaded BLAS libraries matches SuperMatrix since parallelism is abundant within subproblems given sufficiently large problem sizes.

- These results reflect the simplest scheduling of tasks within the SuperMatrix runtime system. Current research investigates sorting the ready and available tasks according to different heuristics to reduce the latency of tasks on the critical path of execution, resulting in better load balance. This sorting of tasks subsumes the concept of compute-ahead [1, 25, 31]. Furthermore, given the modular nature of SuperMatrix, these advanced scheduling techniques are completely abstracted away from the end-user.

Figure 10 shows results for every variant of Cholesky factorization using SuperMatrix and FLAME linked with serial and multithreaded GotoBLAS, respectively. We can clearly see that LAPACK implements Variant 2, which demonstrates that the difference in performance between libFLAME and LAPACK only lies with

the choice of algorithm. All three variants of Cholesky factorization implemented using SuperMatrix share the same performance signature, which provides empirical evidence of the principles explained in Section 5. Even though it is not shown, we have similar results for TRINV and TTMM.

7 Conclusion

Using the FLASH extension to the FLAME/C API, library developers can write algorithms to compute linear algebra operations and leave the details of parallelization to the SuperMatrix runtime system. The high-level abstractions used by SuperMatrix also allow parallelism to be extracted across subroutine boundaries, a task that has proven intractable with conventional methods.

Despite the difficulty in identifying different types of dependencies, SuperMatrix implements and completely abstracts this process from the user. Given any sequential algorithm for computing a linear algebra operation, SuperMatrix can meet or exceed the performance of almost any other algorithm implemented with multithreaded BLAS. This fact frees us from needing to find a family of algorithms, which is often required in order to identify the algorithm with the best possible performance on specific target platforms.

Related work

In [7], all algorithms for each of the three sweeps of SPD-INV were enumerated, as shown in Figure 1, in order to find the best performing algorithmic variant for each sweep. The computation was then rearranged into one sweep, which allowed the authors to apply manual intra-iterational optimizations using PLAPACK [34].

The work in [7] was intended to derive parallelism from the BLAS implementations with some further optimizations applicable only for distributed-memory architectures. By comparison, SuperMatrix runs on shared-memory architectures and exposes parallelism between BLAS subproblems and does so automatically via dependency analysis.

CellSs [4] also applies the ideas of out-of-order execution for thread-level parallelism. The key difference between CellSs and SuperMatrix is the API for identifying tasks. CellSs uses annotations, similar to the compiler directives in OpenMP, which are placed around function calls to denote different tasks. A source-to-source C compiler uses these annotations to insert code that performs the dependency analysis and subsequent out-of-order execution at runtime. The load balancing of tasks can vary greatly according to the computational runtime of each function. On the other hand, the computational runtime of each task in SuperMatrix is bound to the size of each submatrix block created using the FLASH API [27].

Another difference between the current project and CellSs relates to the scheduling of tasks. CellSs uses a scheduling method similar to work-stealing where tasks are assigned to certain threads during the analysis phase but can migrate between threads during execution. SuperMatrix employs a simpler concept without sacrificing flexibility where idle threads dequeue ready and available tasks from a global task queue.

Since SuperMatrix is highly portable, we also have tentative plans to adapt it to the Cell processor.

Additional information

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

Acknowledgments

We thank the other members of the FLAME team for their support.

This research was partially sponsored by NSF grants CCF–0540926 and CCF–0702714.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation (NSF).

References

[1] C. Addison, Y. Ren, and M. van Waveren. OpenMP issues arising in the development of parallel BLAS and LAPACK libraries. *Scientific Programming*, 11(2), 2003.

[2] R. C. Agarwal and F. G. Gustavson. Vector and parallel algorithms for Cholesky factorization on IBM 3090. In *SC '89: Proceedings of the 1989 ACM/IEEE Conference on Supercomputing*, pages 225–233, New York, NY, USA, 1989.

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

[4] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: A programming model

for the Cell BE architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, pages 5–15, Tampa, FL, USA, November 2006.

[5] Paolo Bientinesi. *Mechanical Derivation and Systematic Analysis of Correct Linear Algebra Algorithms*. PhD thesis, The University of Texas at Austin, 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 Transactions on Mathematical Software*, 31(1):1–26, March 2005.

[7] Paolo Bientinesi, Brian Gunter, and Robert van de Geijn. Families of algorithms related to the inversion of a symmetric positive definite matrix. *ACM Transactions on Mathematical Software*. Submitted.

[8] Paolo Bientinesi, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Representing linear algebra algorithms in code: The FLAME application programming interfaces. *ACM Transactions on Mathematical Software*, 31(1):27–59, March 2005.

[9] 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 Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 116–125, San Diego, CA, USA, June 2007.

[10] 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 the 2007 IEEE International Conference on Cluster Computing*, Austin, TX, USA, September 2007.

[11] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

[12] 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 Computer Society Press, 1992.

[13] Timothy Collins and James C. Browne. Matrix++: An object-oriented environment for parallel high-performance matrix computations. In *Proceedings of the Hawaii International Conference on Systems and Software*, 1995.

[14] Timothy Scott Collins. *Efficient Matrix Computations through Hierarchical Type Specifications*. PhD thesis, The University of Texas at Austin, 1996.

[15] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Iain Duff. A set of level 3 Basic Linear Algebra Subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, March 1990.

[16] 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.

[17] Kazushige Goto. `http://www.tacc.utexas.edu/resources/software`.

[18] Kazushige Goto and Robert A. van de Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software*. To appear.

[19] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, December 2001.

[20] F. G. Gustavson, L. Karlsson, and B. Kagstrom. Three algorithms on distributed memory using packed storage. B. Kagstrom, E. Elmroth, editors, *Computational Science – PARA '06, Lecture Notes in Computer Science*. Springer-Verlag, 2007. To appear.

[21] Greg Henry. BLAS based on block data structures. Theory Center Technical Report CTC92TR89, Cornell University, February 1992.

[22] José Ramón Herrero. *A framework for efficient execution of matrix computations*. PhD thesis, Polytechnic University of Catalonia, Spain, 2006.

[23] James Kahle, Michael Day, Peter Hofstee, Charles Johns, Theodore Maeurer, and David Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4/5):589–604, September 2005.

[24] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra. Solving systems of linear equations on the Cell processor using Cholesky factorization. Technical Report UT-CS-07-596, Innovative Computing Laboratory, University of Tennesse, April 2007.

[25] Jakub Kurzak and Jack Dongarra. Implementing linear algebra routines on multi-core processors with pipelining and a look ahead. LAPACK Working Note 178 Technical Report UT-CS-06-581, University of Tennessee, September 2006.

[26] Charles Leiserson and Aske Plaat. Programming parallel applications in Cilk. *SINEWS: SIAM News*, 31, 1998.

[27] Tze Meng Low and Robert van de Geijn. An API for manipulating matrices stored by blocks. FLAME Working Note #12 TR-2004-15, Department of Computer Sciences, The University of Texas at Austin, May 2004.

[28] Bryan Marker, Field G. Van Zee, Kazushige Goto, Gregorio Quintana-Ortí, and Robert A. van de Geijn Toward scalable matrix multiply on multithreaded architectures. In *Euro-Par '07: Proceedings of the Thirteenth International European Conference on Parallel and Distributed Computing*, Rennes, France, August 2007.

[29] N. Park, B. Hong, and V. K. Prasanna. Tiling, block data layout, and memory hierarchy performance. *IEEE Transactions on Parallel and Distributed Systems*, 14(7):640–654, 2003.

[30] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Ernie Chan, Robert A. van de Geijn, and Field G. Van Zee. Scheduling of QR factorization algorithms on SMP and multi-core architectures. FLAME Working Note #24 TR-07-37, Department of Computer Sciences, The University of Texas at Austin, July 2007.

[31] Peter Strazdins. A comparison of lookahead and algorithmic blocking techniques for parallel matrix factorization. *International Journal of Parallel and Distributed Systems and Networks*, 4(1):26–35, June 2001.

[32] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, 11(1), 1967.

[33] 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.

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

[35] Field G. Van Zee, Paolo Bientinesi, Tze Meng Low, and Robert A. van de Geijn. Scalable parallelization of FLAME code via the workqueuing model. *ACM Transactions on Mathematical Software*. To appear.