Satisfying Your Dependencies with SuperMatrix

Ernie Chan [#1], Field G. Van Zee [#2], Enrique S. Quintana-Ortí [*3], Gregorio Quintana-Ortí [*4], Robert van de Geijn [#5]

[#]*Department of Computer Sciences, The University of Texas at Austin*
Austin, Texas, USA
[1]echan@cs.utexas.edu
[2]field@cs.utexas.edu
[5]rvdg@cs.utexas.edu

[*]*Departamento de Ingeniería y Ciencia de Computadores, Universidad Jaume I*
Campus Riu Sec, 12.071, Castellón, Spain
[3]quintana@icc.uji.es
[4]gquintan@icc.uji.es

***Abstract*— SuperMatrix out-of-order scheduling leverages high-level abstractions and straightforward data dependency analysis to provide a general-purpose mechanism for obtaining parallelism from a wide range of linear algebra operations. Viewing submatrices as the fundamental unit of data allows us to decompose operations into component tasks that operate upon these submatrices. Data dependencies between tasks are determined by observing the submatrix blocks read from and written to by each task. We employ the same dynamic out-of-order execution techniques traditionally exploited by modern superscalar micro-architectures to execute tasks in parallel according to data dependencies within linear algebra operations. This paper provides a general explanation of the SuperMatrix implementation followed by empirical evidence of its broad applicability through performance results of several standard linear algebra operations on a wide range of computer architectures.**

I. Introduction

This paper explores the broad applicability of SuperMatrix out-of-order scheduling. In [7] we used storage by blocks [8], [18], [24] to view submatrices as the basic unit of data and tasks that perform operations on those blocks as the basic unit of computation, which results in *algorithms-by-blocks* [2], [9], [10], [17], [19]. After calculating all data dependencies between tasks, dynamic out-of-order execution techniques similar to Tomasulo's algorithm [26] can be used to exploit parallelism within linear algebra operations.

The main contributions of the present paper include:

- Using the simple high-level abstractions [6], [16], [22] presented in [7], we apply SuperMatrix to a wide range of linear algebra operations to exploit parallelism sometimes unattainable by traditional methods [1], [20], [25] without adding additional complexity to the code [27].
- The implementation details of the SuperMatrix mechanism are exposed.
- We provide empirical evidence that the performance of generalized SuperMatrix implementations match or even exceed the performance of linear algebra operations linked with multithreaded BLAS libraries [3], [11] across several different computer architectures.
- The next phase of research to improve performance and expand SuperMatrix functionality is discussed.

The rest of the paper is organized as follows. In Section II we explain the general implementation of SuperMatrix out-of-order scheduling. Section III provides a wide variety of performance graphs. We conclude the paper in Section IV.

II. SuperMatrix Out-of-Order Scheduling

We designed the SuperMatrix mechanism to resemble the inspector–executor method for parallelism [23], [29]. We delay the execution of tasks during the *analyzer* phase to calculate data dependencies. We then execute the tasks in parallel according the explicit data flow specified by their data dependencies during the *scheduler/dispatcher* phase.

A. Analyzer

Currently the SuperMatrix mechanism assumes that the input matrices are stored hierarchically with one level of blocking where submatrices are square. We provide the FLASH API [22] to create and access these hierarchical matrices since users do not need to know the underlying storage of matrices. Using this API, users create a matrix of matrices where each element in the top level matrix is a pointer to another matrix. Given this level of indirection, the SuperMatrix mechanism appends information detailing the tasks that read from and write to each block.

As each task is enqueued onto the task queue in sequential program order, each submatrix structure tracks the tasks that read from and write to its data in order to compute flow, anti, and output data dependencies between all tasks. This data dependency information is stored explicitly within the SuperMatrix task structure.

In Fig. 1 (left), we present the SuperMatrix implementation of LU factorization without pivoting. The calls to `FLASH_LU_nopiv`, `FLASH_Trsm`, and `FLASH_Gemm` decompose themselves into component tasks operating on square blocks and then place those tasks onto the task queue. In a corresponding sequential implementation, those routines would execute the operations without delay.

B. Scheduler/Dispatcher

Once all the tasks are enqueued onto the task queue, the call to `FLASH_Queue_exec` initiates the parallel execution,

```
FLA_Error FLASH_LU_nopiv_var5( 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_Obj_width ( ATL ) < FLA_Obj_width ( A ) )
   {
      FLA_Repart_2x2_to_3x3( ATL, /**/ ATR,       &A00, /**/ &A01, &A02,
                           /* ************* */   /* ******************** */
                                                  &A10, /**/ &A11, &A12,
                             ABL, /**/ ABR,       &A20, /**/ &A21, &A22,
                             1, 1, FLA_BR );

      /*-----------------------------------------------------------------*/

      FLASH_LU_nopiv( A11 );

      FLASH_Trsm( FLA_LEFT, FLA_LOWER_TRIANGULAR,
                  FLA_NO_TRANSPOSE, FLA_UNIT_DIAG,
                  FLA_ONE, A11, A12 );

      FLASH_Trsm( FLA_RIGHT, FLA_UPPER_TRIANGULAR,
                  FLA_NO_TRANSPOSE, FLA_NONUNIT_DIAG,
                  FLA_ONE, A11, A21 );

      FLASH_Gemm( FLA_NO_TRANSPOSE, FLA_NO_TRANSPOSE,
                  FLA_MINUS_ONE, A21, A12, FLA_ONE, A22 );

      /*-----------------------------------------------------------------*/

      FLA_Cont_with_3x3_to_2x2( &ATL, /**/ &ATR,       A00, A01, /**/ A02,
                                                       A10, A11, /**/ A12,
                              /* ************* */   /* ****************** */
                                &ABL, /**/ &ABR,       A20, A21, /**/ A22,
                                FLA_TL );
   }
   FLASH_Queue_exec( );

   return FLA_SUCCESS;
}
```

Stage	Scheduled Tasks			
1	LU			
2	TRSM	TRSM	TRSM	TRSM
3	TRSM	TRSM	TRSM	TRSM
4	GEMM	GEMM	GEMM	GEMM
5	GEMM	GEMM	GEMM	GEMM
6	GEMM	GEMM	GEMM	GEMM
7	GEMM	GEMM	GEMM	GEMM
8	LU			
9	TRSM	TRSM	TRSM	TRSM
10	TRSM	TRSM	GEMM	GEMM
11	GEMM	GEMM	GEMM	GEMM
12	GEMM	GEMM	GEMM	LU
13	TRSM	TRSM	TRSM	TRSM
14	GEMM	GEMM	GEMM	GEMM
15	LU			
16	TRSM	TRSM		
17	GEMM			
18	LU			

Fig. 1. Left: SuperMatrix implementation of LU factorization without pivoting. Right: Simulated SuperMatrix execution of LU factorization without pivoting on a 5×5 matrix of blocks using 4 threads. Each column represents the tasks executed on separate threads.

which we implemented using a global FIFO waiting queue separate from the task queue.

Once a task completes execution, all dependent tasks that use blocks updated by the recently completed task are notified. If a notified task has all of its dependencies fulfilled, it is marked as ready and available. Those ready and available tasks are enqueued at the tail of the waiting queue while idle threads dequeue tasks from the head of the waiting queue until all tasks have been executed. To begin execution, it is an invariant that the first task in the task queue is always ready and available.

We used OpenMP to provide multithreading facilities where each thread executes asynchronously. We have also implemented SuperMatrix using the POSIX threads API to reach a broader range of platforms.

In Fig. 1 (right), we show the simulated SuperMatrix execution of LU factorization without pivoting on a 5×5 matrix of blocks using 4 threads. Each column represents tasks that execute on separate threads. This simulation assumes that each thread performs lock-step synchronization after executing a single task, which we denote as a single stage, for illustration purposes. The SuperMatrix mechanism uses asynchronous threads via fine grained locking.

In the simulation, we can see the tasks from `FLASH_Trsm` and `FLASH_Gemm` are interleaved in stage 10. Those tasks represent intra-iterational parallelism since those operations lie within the main `while` loop in Fig. 1 (left). Stage 12 illustrates inter-iterational parallelism since a task from `FLASH_LU_nopiv` is interleaved with tasks from `FLASH_Gemm` in the previous iteration.

SuperMatrix out-of-order scheduling derives its parallelism from the data flow specified by data dependencies within the operation. No other information specific to the linear algebra operations is used to exploit parallelism.

Architecture	PEs	Peak (GFLOPS)	BLAS
Itanium2	16	96.0	MKL 8.1
Xeon[1]	8	41.6	MKL 9.0
Opteron	8	41.6	ACML 3.6
POWER5	8	60.8	ESSL 4.2

TABLE I

ARCHITECTURES ON WHICH THE LINEAR ALGEBRA OPERATIONS ARE PERFORMED. NOTE THAT ALL THESE ARCHITECTURES ALSO HAVE GOTOBLAS 1.13 INSTALLED.

BLAS			LAPACK
GEMM	HEMM	SYMM	CHOL
TRMM	HER2K	SYR2K	LU
TRSM	HERK	SYRK	SPD-INV

TABLE II

THE LEVEL-3 BLAS AND LAPACK-LIKE OPERATIONS SUPPORTED BY SUPERMATRIX. NOTE THAT WE HAVE LU FACTORIZATION *without pivoting*.

III. PERFORMANCE

In this section, we provide empirical evidence of the wide applicability of the SuperMatrix mechanism for several linear algebra operations across different computer architectures.

[1]Four dual-core "Tulsa" processors

A. Target architectures and supported operations

We performed experiments on the four different architectures listed in Table I where each has at least eight processing elements (PEs). A processing element is a single core in multicore processors or a CPU in SMP systems. Peak performance is listed in gigaflops/sec (GFLOPS). We installed the GotoBLAS 1.13 library [12] on each machine and also had access to the commercial vendors' BLAS libraries.

Table II lists the linear algebra operations that the SuperMatrix mechanism currently supports, including all the level-3 BLAS operations along with several LAPACK-like operations. The implementations of these operations took only a few days to program thanks to the relative simplicity of the SuperMatrix interface.

Inversion of a symmetric positive definite matrix (SPD-INV) is one LAPACK operation composed of several other LAPACK operations, which can be implemented by Cholesky factorization followed by inversion of a triangular matrix and then triangular matrix multiplication by its transpose [5].

B. Implementations

In this section, we provide the results for a subset of the supported operations: Cholesky factorization (CHOL), LU factorization without pivoting (LU), general matrix-matrix multiplication (GEMM), symmetric rank-k update (SYRK), triangular matrix-matrix multiplication (TRMM), and triangular solve with multiple right-hand sides (TRSM).

We used an algorithmic block size $b = 192$ for all experiments and created problem instances of CHOL, LU, TRMM, and TRSM where all input matrices are square. We performed GEMM, $C = AB + C$, where C is $m \times m$, A is $m \times k$, B is $k \times m$, and k is fixed to b, which is indicative of the instances of GEMM typically encountered in practice. We also fixed the k dimension of SYRK to b in similar fashion.

The LAPACK-like operations CHOL and LU have three separate implementations:

- **SuperMatrix + serial BLAS**
  The SuperMatrix implementation of CHOL is similar to the one for LU in Fig. 1 (left). For the execution of individual tasks on each thread, we linked to serial BLAS libraries.
  The results are reflective of our simplest SuperMatrix mechanism. We did not use advanced scheduling techniques such as *data affinity* described in [7].
- **FLAME + multithreaded BLAS**
  We linked the sequential implementations of `FLA_Chol` and `FLA_LU_nopiv` provided by libFLAME 1.0 with multithreaded BLAS libraries.
  `FLA_Chol` implements the right-looking algorithm, which is rich with rank-k updates that are straightfoward to parallelize. `FLA_LU_nopiv` implements the right-looking algorithm, which is the blocked variant of classic Gaussian elimination.
- **LAPACK + multithreaded BLAS**
  LAPACK 3.0 only provides LU factorization *with pivoting* as `dgetrf`. We edited that routine to eliminate pivoting and named it `dgetnf`, hence the label **LAPACK-like**.
  We linked LAPACK's implementation of CHOL, `dpotrf`, and `dgetnf` with multithreaded BLAS libraries. We also modified both routines to use an algorithmic block size of 192 instead of the block sizes provided by LAPACK's `ilaenv` routine.
  `dpotrf` implements the left-looking algorithm, which is rich in matrix-panel multiplication that does not parallelize well. `dgetnf`, like `dgetrf`, implements the right-looking algorithm.

The level-3 BLAS operations GEMM, SYRK, TRMM, and TRSM have two implementations: **SuperMatrix + serial BLAS** and **multithreaded BLAS**.

C. Results

Fig. 2 through Fig. 5 present the results for each of the four architectures each using eight processing elements.

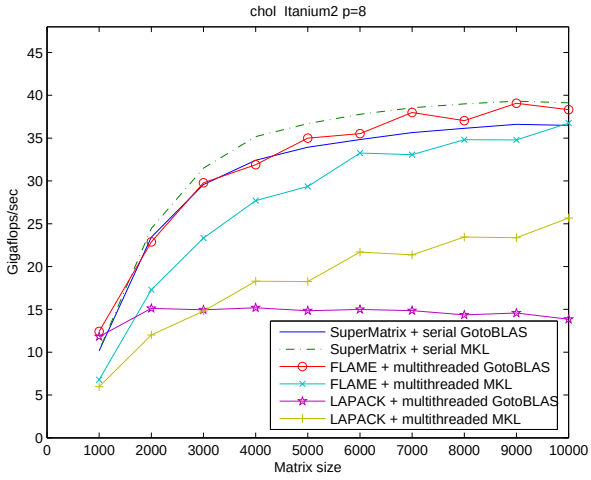
Since the level-3 BLAS operations are the main components comprising both CHOL and LU, a few comments are due:

- GEMM and SYRK have no data dependencies between their component tasks. SuperMatrix must still incur the cost of finding potential data dependencies between tasks despite GEMM and SYRK exhibiting abundant parallelism. TRMM and TRSM on the other hand have a few data dependencies between their component tasks.
- In order to get high performance, BLAS libraries pack matrices into buffers to obtain stride one access during execution [13]. Given large matrices, the cost of packing is amortized over the computation. Since blocks are generally accessed by several tasks, those blocks must be repeatedly packed and unpacked during the execution of each task for SuperMatrix.
- Despite these performance penalties, SuperMatrix remains competitive for these operations compared to the multithreaded implementations provided by architecture-tuned BLAS libraries.

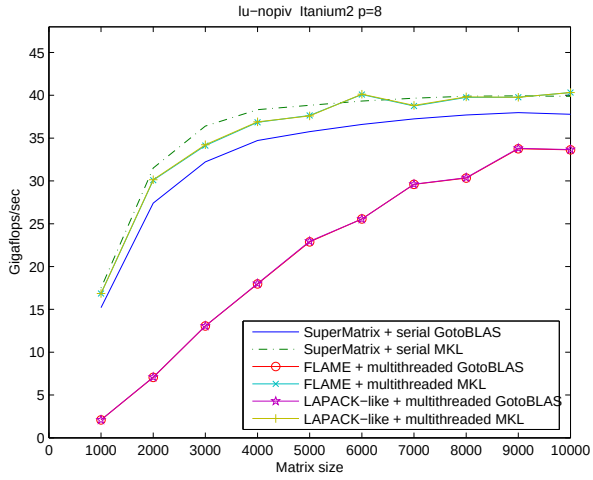
These results are also indicative of the comparative performance attained by the other level-3 BLAS operations.

Because the level-3 BLAS operations have few, if any, data dependencies, SuperMatrix can potentially extract parallelism at a finer granularity for LAPACK-like operations than the traditional sequential algorithms linked to multithreaded BLAS libraries. These results deserve several comments:

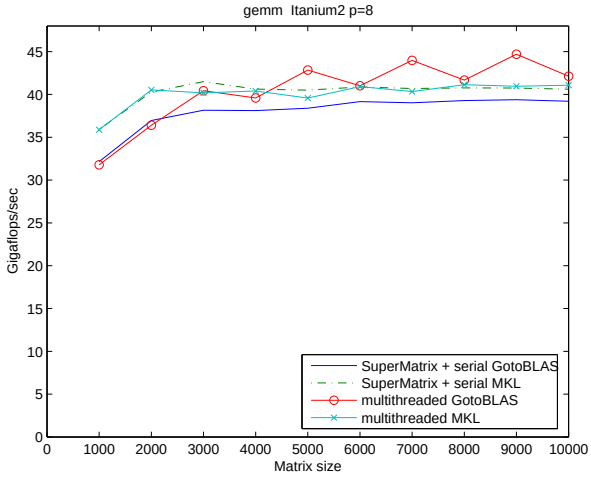
- The SYRK implementation in GotoBLAS typically performs best when the k dimension is fixed. This instance of SYRK is used in the right-looking algorithm for CHOL, which gives rise to `FLA_Chol` performing best when linked to multithreaded GotoBLAS.
- In Fig. 4 (a) `FLA_Chol` performs poorly since ACML performance is nearly sequential for the instance of SYRK shown in Fig. 4 (d).
- Despite our tuning the block size, `dpotrf` does not perform as well as `FLA_Chol` when linked with multithreaded BLAS. LAPACK uses an algorithmic variant



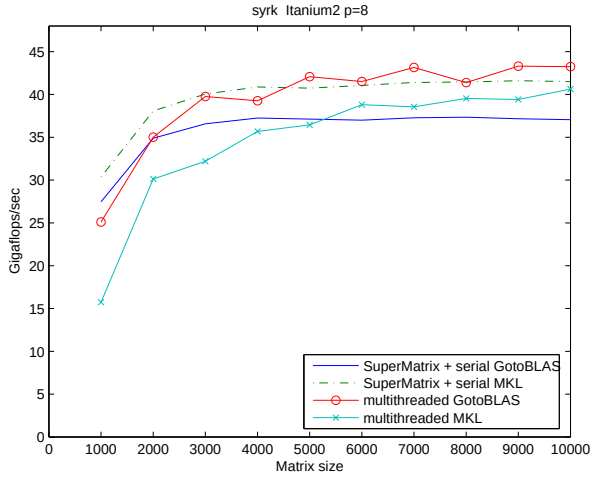
(a)



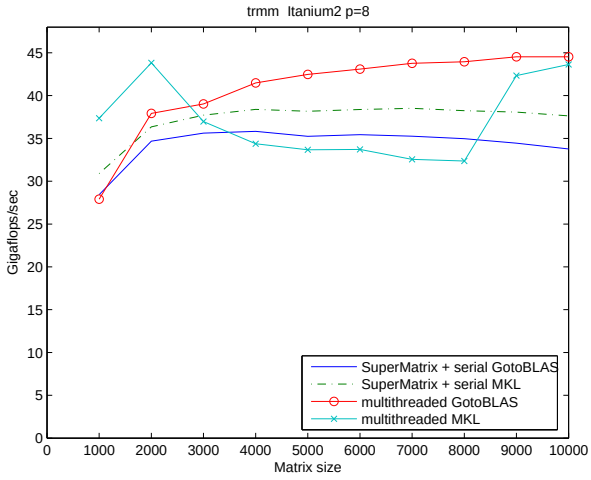
(b)



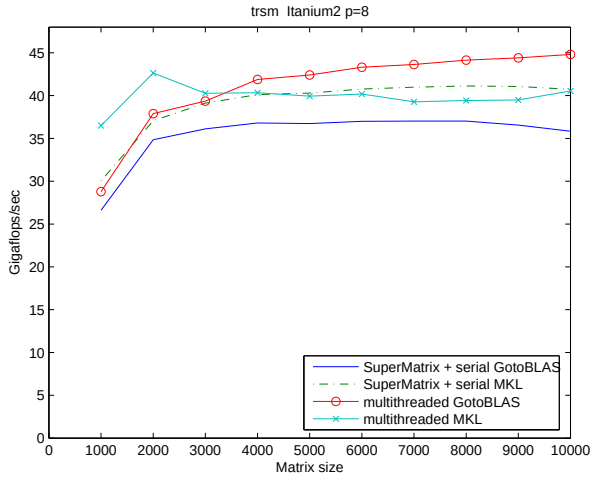
(c)



(d)

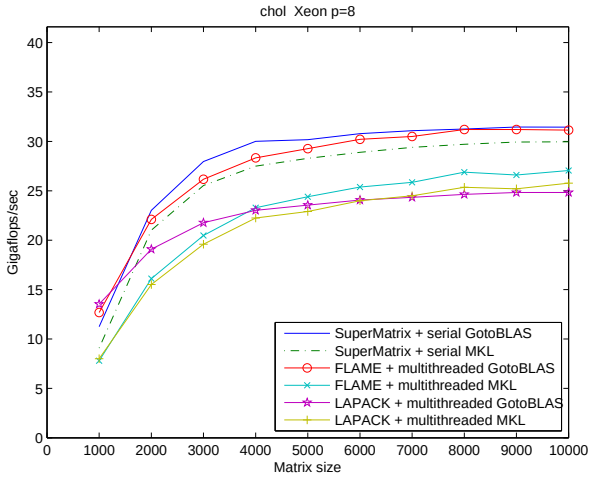


(e)

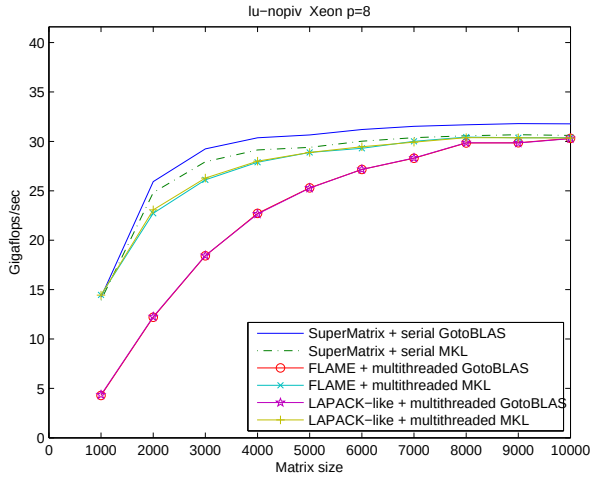


(f)

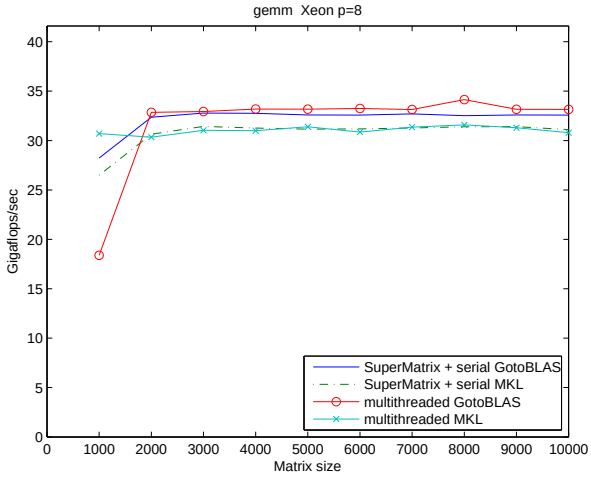
Fig. 2. Results from the Itanium2 machine.



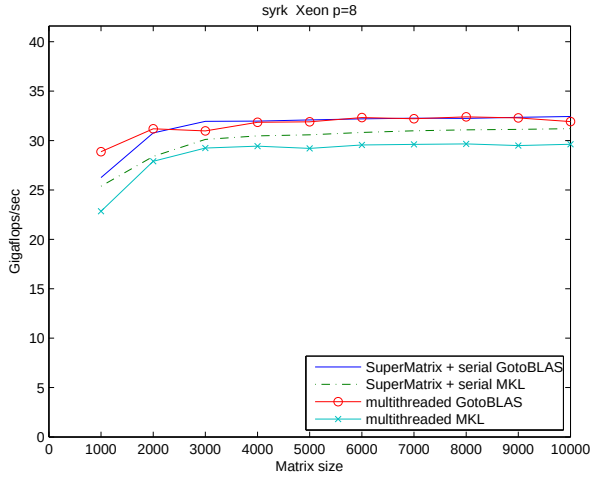
(a)



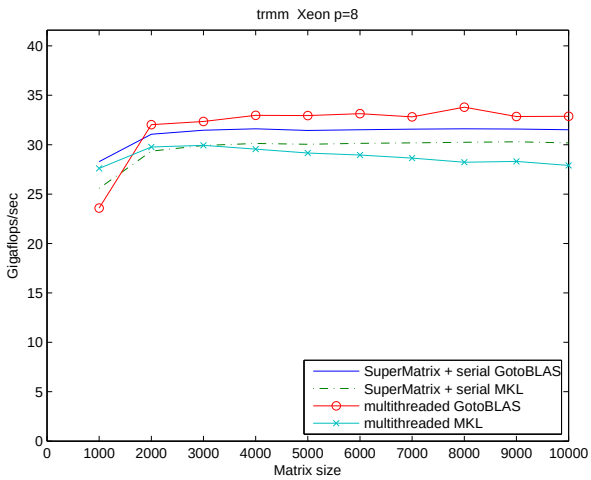
(b)



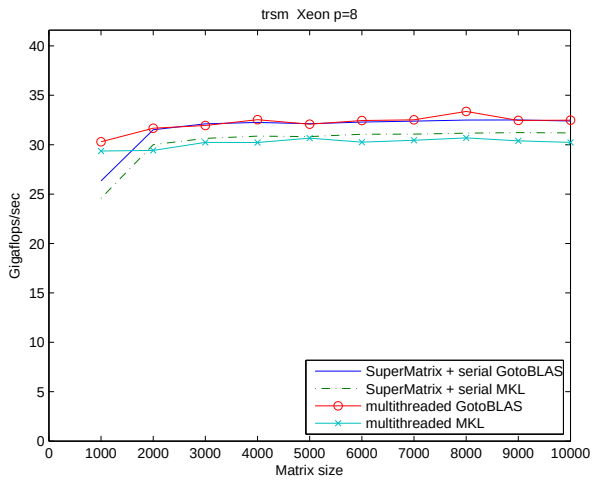
(c)



(d)

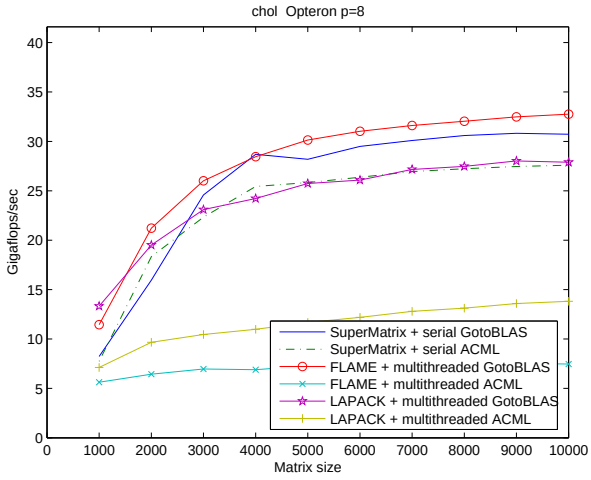


(e)

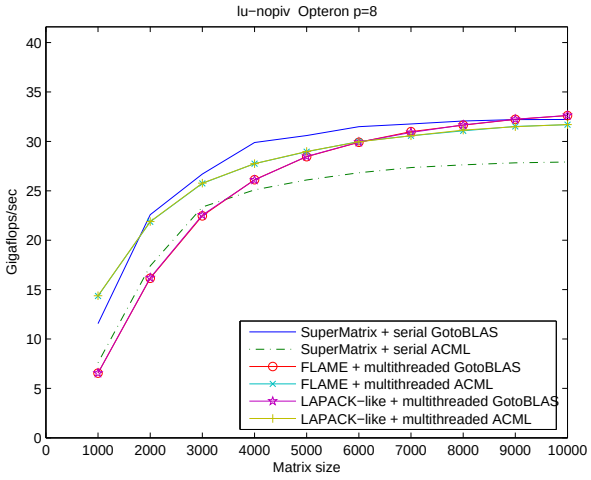


(f)

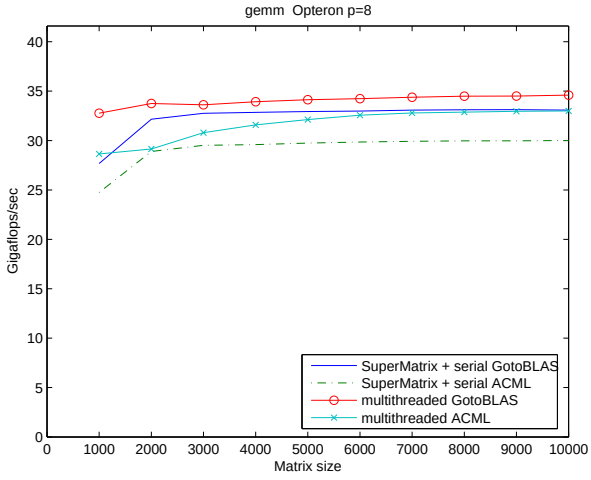
Fig. 3. Results from the Xeon machine.



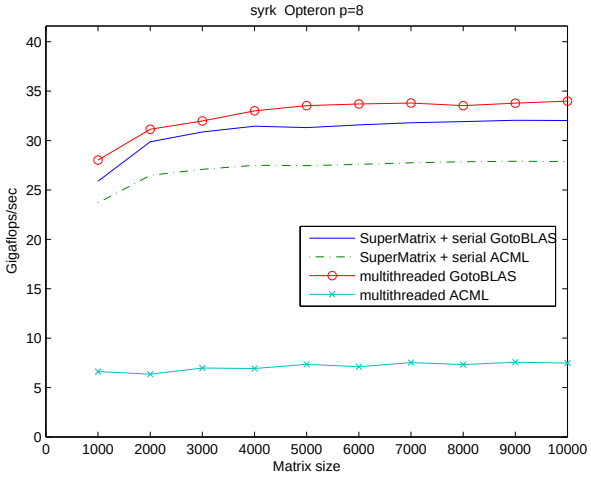
(a)



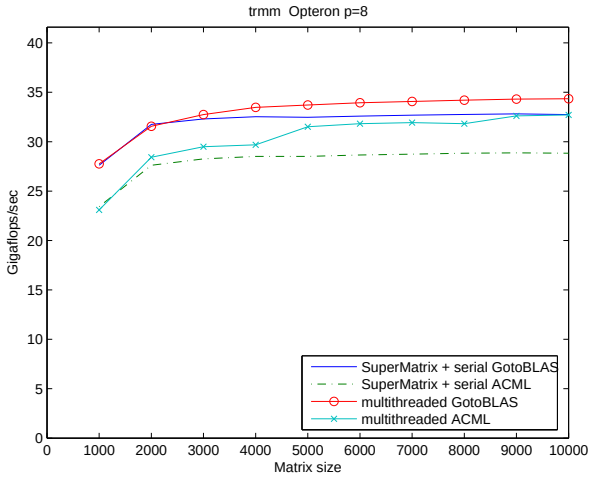
(b)



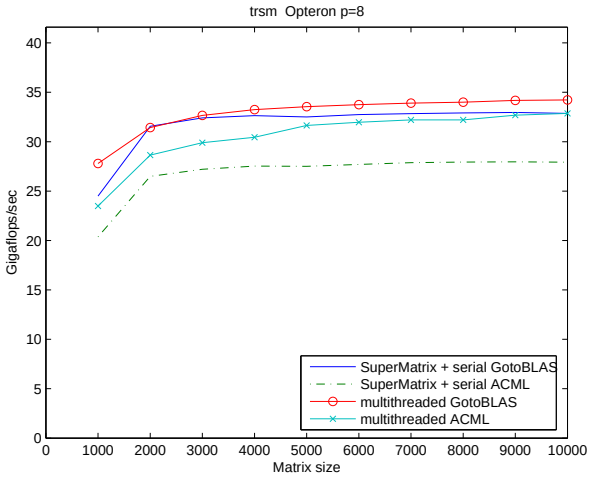
(c)



(d)

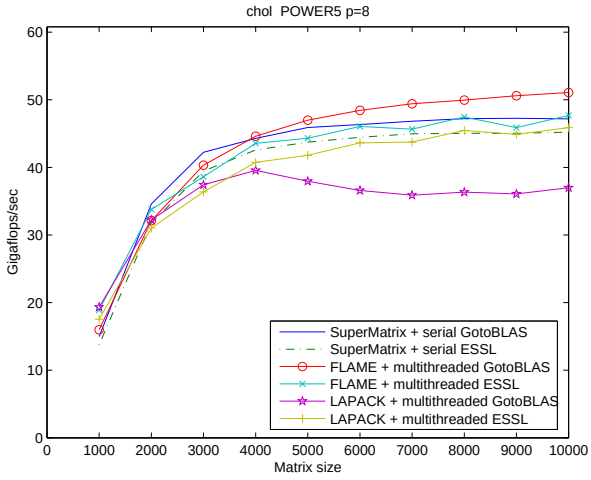


(e)

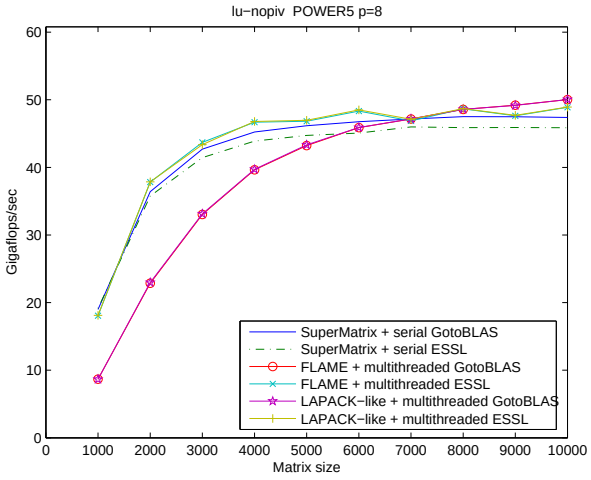


(f)

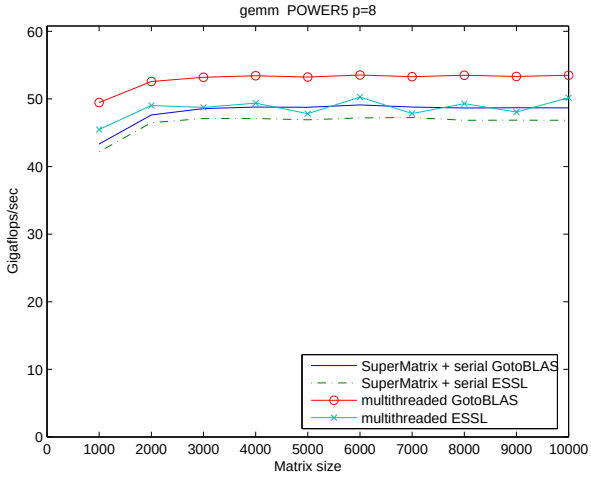
Fig. 4. Results from the Opteron machine.



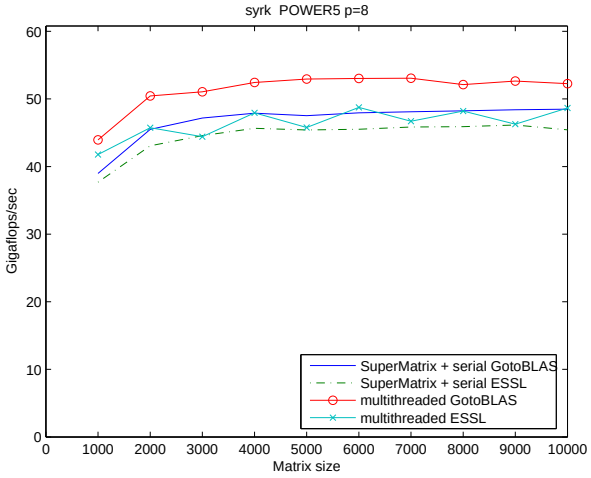
(a)



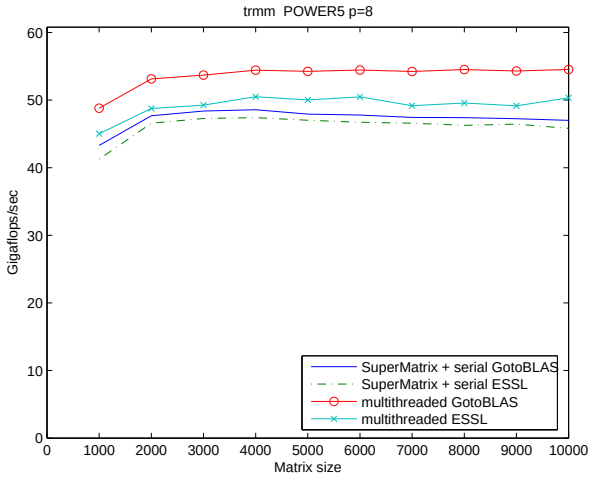
(b)



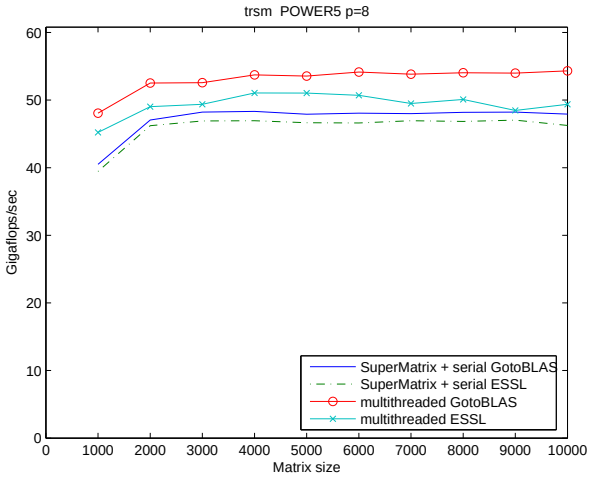
(c)



(d)



(e)



(f)

Fig. 5. Results from the POWER5 machine.

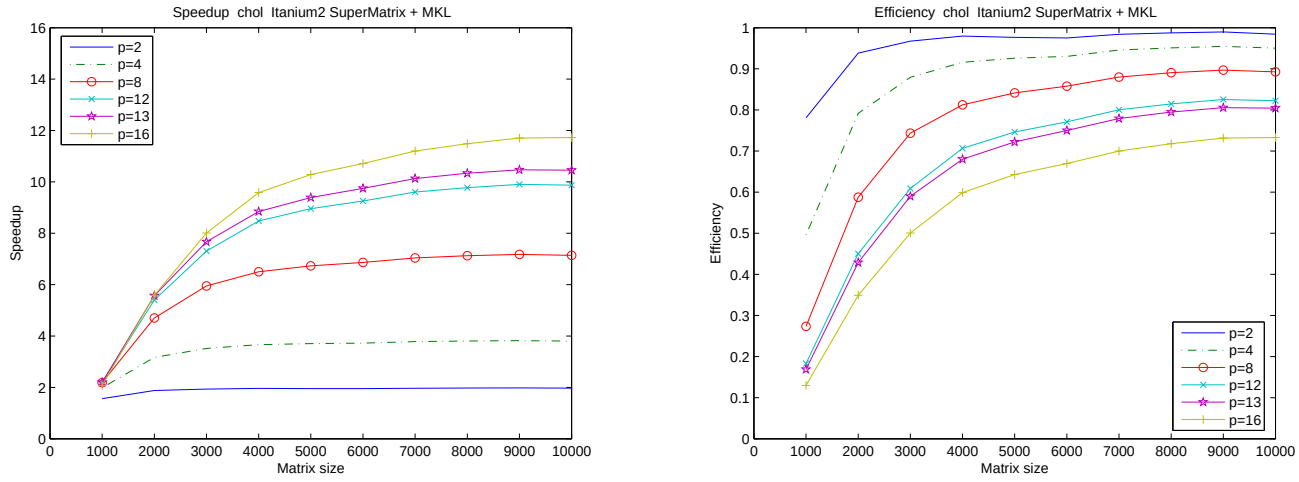


Fig. 6. Speedup and efficiency of the Cholesky factorization implemented with SuperMatrix linked with MKL on the Itanium2 machine with 16 processors.

that is rich in matrix-panel multiply, which does not parallelize as well as the SYRK based variant used by libFLAME.

- `FLA_LU_nopiv` and `dgetnf` implement the same algorithm, so their performances match closely.
- `FLA_LU_nopiv` and `dgetnf` linked to multithreaded GotoBLAS perform poorly compared to being linked to the different vendors' multithreaded BLAS libraries. The second instance of TRSM called by the right-looking algorithm is not a commonly utilized problem instance, so GotoBLAS does not highly optimize that problem instance of TRSM. When performing LU factorization *with pivoting*, the pivoting occurs in the factorization of the column panel, so the simple call to TRSM is not used in that case.
  We notified Kazushige Goto about the poor performance of this problem instance of TRSM. Subsequent releases of the GotoBLAS library have remedied this deficiency.
- SuperMatrix generally is one of the best performing implementations for CHOL and LU because of its ability to exploit parallelism between operations. The sequential algorithms have implicit synchronization points between each call to multithreaded BLAS operations.

D. Scalability

In Fig. 6 we show speedup and efficiency results for the SuperMatrix implementation of CHOL linked with serial MKL using 2, 4, 8, 12, 13, and 16 threads. We used `FLA_Chol` linked with serial MKL as the baseline for these scalability experiments.

The *analyzer* phase only scales with the problem size while remaining constant when varying the number of threads given its sequential execution, which clearly demonstrates Amdahl's Law. Inherent data dependencies in CHOL also limit the amount of parallelism exploitable by the *scheduler/dispatcher* phase. Both of these issues lead to the degradation in efficiency shown Fig. 6 (right) as we scale up the number of threads. Despite these limitations, SuperMatrix still attains a speedup of approximately 12 when using 16 threads.

IV. Conclusion

The ideas behind SuperMatrix rely on high-level abstractions to expose fine grain parallelism in linear algebra operations. By viewing matrix blocks as the fundamental unit of computation, we can schedule tasks that access and update those blocks using the same out-of-order execution techniques long exploited by modern computer micro-architectures.

SuperMatrix upholds that the whole is greater than the sum of the parts. Despite all the efforts to support a wide variety of multithreaded BLAS operations across multiple computer architectures, solely focusing on parallelizing individual component operations has its fundamental limits. SuperMatrix can match or exceed the performance of these libraries because of its ability to exploit parallelism between operations while being highly portable since it only depends upon a few kernels that perform well sequentially on small square matrices.

Future work

Since we use LU factorization without pivoting as the primary example in this paper, our next phase of research entails input driven control flow such as pivoting strategies. SuperMatrix scheduling derives parallelism from dynamic yet explicit data flow, but pivoting introduces nondeterministic behavior. We gained experience dealing with pivoting strategies from parallelism on distributed-memory architectures [14], [28] and believe the same solutions are applicable.

The SuperMatrix implementations in this paper reflect our initial efforts. Current research involves sorting the ready and available tasks according to different heuristics to attain better load balance between processing elements by reducing the latency of tasks on the critical path of execution. This sorting of tasks subsumes the concept of compute-ahead [1], [20], [25]. Given the modular nature of the SuperMatrix mechanism, these advanced scheduling techniques are completely abstracted away from the user.

We have also begun experiments to compare the SuperMatrix mechanism with linear algebra operations implemented using Cilk [21], a multithreaded programming environment.

Since SuperMatrix is highly portable, we have tentative plans to adapt it to the Cell processor [4], [15], which poses specific challenges due to the explicit management of memory.

Additional information

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

ACKNOWLEDGMENT

We thank Kazushige Goto, Kent Milfeld, and all the staff at the Texas Advanced Computing Center (TACC) for their help and access to their machines: `p4xeon`, `cowbell`, and `champion`. We also 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, 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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

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

[14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI*. The MIT Press, 1994.

[15] Michael Gschwind, H. Peter Hofstee, Brian Flachs, Martin Hopkins, Yukio Watanabe, and Takeshi Yamazaki. Synergistic processing in Cell's multicore architecture. *IEEE Micro*, 26(2):10–24, 2006.

[16] 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.

[17] 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.

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

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

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

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

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

[23] Honghui Lu, Alan L. Cox, Sandhya Dwarkadas, Ramakrishnan Rajamony, and Willy Zwaenepoel. Compiler and software distributed shared memory support for irregular applications. In *PPOPP '97: Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 48–56, New York, NY, USA, 1997.

[24] 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.

[25] 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.

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

[27] 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.

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

[29] Reinhard von Hanxleden, Ken Kennedy, Charles H. Koelbel, Raja Das, and Joel H. Saltz. Compiler analysis for irregular problems in Fortran D. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757, pages 97–111, New Haven, CT, USA, 1992.