GEMM–Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark

Bo Kågström * Per Ling* Charles Van Loan †

October 1995

Abstract

The level 3 Basic Linear Algebra Subprograms (BLAS) are designed to perform various matrix multiply and triangular system solving computations. The development of optimal level 3 BLAS code is costly and time consuming, because it requires assembly level programming/thinking. However, it is possible to develop a portable and high-performance level 3 BLAS library mainly relying on a highly optimized _GEMM, the routine for the general matrix multiply and add operation. With suitable partitioning, all the other level 3 BLAS can be defined in terms of _GEMM and a small amount of level 1 and level 2 computations. Our contribution is two-fold. First, the model implementations in Fortran 77 of the GEMM-based level 3 BLAS, which are structured to effectively reduce data traffic in a memory hierarchy. Second, the GEMM-based level 3 BLAS performance evaluation benchmark, which is a tool for evaluating and comparing different implementations of the level 3 BLAS with the GEMM-based model implementations.

1 Introduction

The memory organization in current advanced computer architectures is hierarchical. Accesses to data in the upper levels of the memory hierarchy (registers, cache and/or local memory) are much faster than those in lower levels (off-processor and shared memory). Typically, the peak performance measured in Mflops or Gflops (10^6 and 10^9 floating point operations per second, respectively) is only delivered for data stored in the top level of the memory hierarchy. Therefore, it is important to organize the computations such that we can maximize reuse of data in the upper levels of the memory hierarchy. Matrix and vector operations are basic in most scientific computations and can most often be reorganized for effective data reuse. For example, an n-by-n matrix

*Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden. Email addresses: bokg@cs.umu.se and per.ling@cs.umu.se. Financial support has been received from the Swedish National Board of Industrial and Technical Development under grant NUTEK 89-02578P.

†Department of Computer Science, Cornell University, Ithaca, New York 14853-7501. Email address: cv@cs.cornell.edu.

Also appears as Umeå University Report UMINF-95.18.

multiply $C = AB$ involves $O(n^3)$ arithmetic but $O(n^2)$ data movements (usually). Thus, as n grows the cost of accessing data is increasingly dominated by the cost of computation. This fact has lead to a technique to reorganize standard algorithms to perform matrix-matrix (level 3) operations in their inner loops (e.g., see [12]). Typically, these matrix-matrix operations are expressed as calls to level 3 Basic Linear Algebra Subprograms (BLAS) [9, 10], which together with level 1 BLAS [22] and level 2 BLAS [7] are de facto standards for basic matrix and vector operations. The level 3 BLAS have been successfully used as building blocks for several applications, including the software library LAPACK [3]. With a highly optimized level 3 BLAS, most of the LAPACK codes will "automatically" peform well.

However, due to the complex hardware organization of advanced computer architectures it can be very costly and time consuming to develop a high-performance level 3 BLAS because it requires assembly level programming/thinking. The GEMM-based approach presents a way to attain high performance and portability with a limited effort. The GEMM-based level 3 BLAS concept [21] shows that it is possible to formulate the level 3 BLAS operations in terms of the level 3 operation for general matrix multiply and add (GEMM) and some level 1 and level 2 BLAS operations. Whenever new high-performance architectures or extensions and modifications of existing ones are introduced, we see the great benefits of the GEMM-approach, since we only require a few underlying routines to be optimized for the target architecture. Most important are the routines that implement the level 3 GEMM operation and the level 2 operation for general matrix-vector multiply and add (GEMV). Morover, the GEMM-based approach provides possibilities to invoke parallelism, for example, by using parallel versions of the underlying routines. It is also possible to create a level 3 BLAS library based on fast algorithms for the GEMM operation, e.g., Strassen's or Winograd's algorithms [25, 26, 15, 11].

Our contribution is two-fold. First, the model implementations in Fortran 77 of the GEMM-based level 3 BLAS, which are structured to effectively reduce data traffic in a memory hierarchy. Second, the GEMM-based level 3 BLAS performance evaluation benchmark, which is a tool for evaluating and comparing different implementations of the level 3 BLAS with the GEMM-based model implementations. All software come in all four data precisions and are designed to be easy to implement and use on different platforms. Each of the GEMM-based routines has a few system dependent parameters that specify internal block sizes, cache characteristics, and intersection points for alternative code sections, which are given as input to a program that facilitates the tuning of these parameters. For simplicity, we also provide sample values for some common architectures.

We present the GEMM-based model implementations and benchmark in a two-part paper. In this part, we review the GEMM-based concept, and we present the design principles behind and the model implementations, and the performance evaluation benchmark. Moreover, we report results from extensive testings on several high-performance platforms. In a companion paper [20], we describe the installation and tuning of the GEMM-based model implementations, and the use and installation of the performance evaluation benchmark.

Before we go into any further details we outline the content of this paper. To set the scene Section 2 gives a brief summary of the level 3 BLAS operations and calling sequences with parameter lists. In Section 3 we summarize the GEMM-based level 3 BLAS concept and illustrate our approach with a sample level 3 BLAS operation. Section 4 discusses the design principles used in the GEMM-based model implementations. In Section 5 we give a more detailed presentation of the high-performance and portable model implementations of the GEMM-based level 3 BLAS. Section 6 presents the GEMM-based level 3 BLAS benchmark, its purpose and design. In Section 7 we report results from our extensive testings. Section 7.1 presents measured performance results for different architectures (vector as well as RISC-based), including single processor results for IBM SP2, Intel Paragon, NEC SX-3, Parsytec GC/PowerPlus and the workstations IBM RS6000 and Silicon Graphics Indy. Notice that any implementation of distributed versions of level 3 operations or block-partitioned algorithms should be designed to minimize communication overhead, as well as to make use of highly optimized single processor level 3 kernels. For an increasing number of processors, any improvement of the performance of a single processor will (at least for a single program multiple data (SPMD) application) have a multiplicative effect on the overall performance. In section 7.2 we present some benchmark results for different level 3 BLAS implementations. Section 8 presents some additional techniques that can be used to even gain some more performance from the GEMM-based model implementations. Finally, in Section 9 we give some conclusions regarding our contributions.

2 Level 3 BLAS Summary

The level 3 BLAS consist of routines for both general and "structured" matrix multiplication, including multiple right hand side triangular system solving. The six level 3 operations (and routine names) are general matrix multiply and add (_GEMM), symmetric matrix multiply (_SYMM, _HEMM), symmetric rank-k update (_SYRK, _HERK), symmetric rank-$2k$ update (_SYR2K, _HER2K), triangular matrix multiply (_TRMM), and triangular system solve (_TRSM). In a complete implementation of the level 3 BLAS there are four versions of the routines _GEMM, _SYMM, _SYRK, _SYR2K, _TRMM and _TRSM corresponding to four different data types and with the following prefixes of the routine names: S for single precision real data, D for double precision real data, C for single precision complex data and Z for double precision complex data. The routines _HEMM, _HERK and _HER2K concern hermitian matrices and therefore they only exist with prefixes C and Z.

In tables 1 and 2, the operations and the parameter lists of the level 3 BLAS are summarized [3]. We mainly discuss the real case here. Some comments on the complex case are given in Section 5.6.

Options of a level 3 operation are controlled by arguments that specify $\text{op}(X) = X, X^T$ or X^H (`trans` = `'N'`, `'T'` or `'C'`), the storage format of $\text{op}(X)$; upper or lower triangular (`uplo` = `'U'` or `'L'`), unit or non-unit triangular (`diag` = `'U'` or `'N'`), and whether $\text{op}(X)$ should be applied from left or right (`side` = `'L'` or `'R'`). Each routine has two or four (only _TRMM and _TRSM) form parameters for specifying different options. The parameters `m`, `n` and `k` specify the sizes of the matrices A, B and/or C

Table 1: Level 3 BLAS operations.

Routine	Operation	Prefixes
_GEMM	$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C, \mathrm{op}(X) = X, X^T, X^H, C$ is $m \times n$	S, D, C, Z
_SYMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C$ is $m \times n, A = A^T$	S, D, C, Z
_HEMM	$C \leftarrow \alpha AB + \beta C, C \leftarrow \alpha BA + \beta C, C$ is $m \times n, A = A^H$	C, Z
_SYRK	$C \leftarrow \alpha AA^T + \beta C, C \leftarrow \alpha A^T A + \beta C, C = C^T$ is $n \times n$	S, D, C, Z
_HERK	$C \leftarrow \alpha AA^H + \beta C, C \leftarrow \alpha A^H A + \beta C, C = C^H$ is $n \times n$	C, Z
_SYR2K	$C \leftarrow \alpha AB^T + \alpha BA^T + \beta C, C \leftarrow \alpha A^T B + \alpha B^T A + \beta C, C = C^T$ is $n \times n$	S, D, C, Z
_HER2K	$C \leftarrow \alpha AB^H + \alpha BA^H + \beta C, C \leftarrow \alpha A^H B + \alpha B^H A + \beta C, C = C^H$ is $n \times n$	C, Z
_TRMM	$C \leftarrow \alpha \mathrm{op}(A)C, C \leftarrow \alpha C\mathrm{op}(A), \mathrm{op}(A) = A, A^T, A^H, C$ is $m \times n$	S, D, C, Z
_TRSM	$C \leftarrow \alpha \mathrm{op}(A^{-1})C, C \leftarrow \alpha C\mathrm{op}(A^{-1}), \mathrm{op}(A) = A, A^T, A^H, C$ is $m \times n$	S, D, C, Z

that participate in the operations, whose leading dimensions are specified by `lda`, `ldb` and `ldc`, respectively. Finally, `alpha` and `beta` correspond to the scalars α and β in the operations.

Table 2: Level 3 BLAS parameter lists.

Routine	Parameters
_GEMM	`( transa, transb, m, n, k, alpha, A, lda, B, ldb, beta, C, ldc )`
_SYMM	`( side, uplo, m, n, alpha, A, lda, B, ldb, beta, C, ldc )`
_HEMM	`( side, uplo, m, n, alpha, A, lda, B, ldb, beta, C, ldc )`
_SYRK	`( uplo, trans, n, k, alpha, A, lda, beta, C, ldc )`
_HERK	`( uplo, trans, n, k, alpha, A, lda, beta, C, ldc )`
_SYR2K	`( uplo, trans, n, k, alpha, A, lda, B, ldb, beta, C, ldc )`
_HER2K	`( uplo, trans, n, k, alpha, A, lda, B, ldb, beta, C, ldc )`
_TRMM	`( side, uplo, trans, diag, m, n, alpha, A, lda, C, ldc )`
_TRSM	`( side, uplo, trans, diag, m, n, alpha, A, lda, C, ldc )`

3 GEMM–Based Level 3 BLAS Concept

We have shown that "one can live with" just one highly optimized level 3 BLAS routine: _GEMM [21, 17]. This subprogram oversees a general matrix multiply of the form

$$C \leftarrow \alpha \mathrm{op}(A)\mathrm{op}(B) + \beta C, \quad \text{where} \quad \mathrm{op}(X) \quad \text{denotes} \quad X \text{ or } X^T.$$

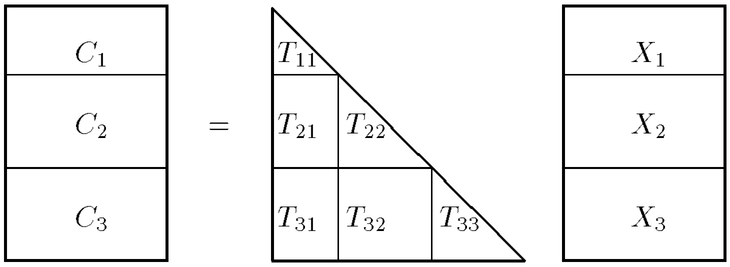
The structured matrix multiplication problems handled by the other level 3 BLAS can be couched in terms of _GEMM and a small amount of level 1 and 2 computations, with suitable partitionings. Roughly, the idea is to reduce the overall structured multiplication to a set of general multiplications involving "strips". Here, a strip is either a block row or a block column. For performance purposes a strip can be further partitioned into subblocks. We illustrate our approach with the operation _TRSM for triangular system solve with multiple right hand sides.

From tables 1 and 2, we know the operations and the parameters of _TRSM. The matrix A is triangular and the parameters `side` and `trans` are used to specify its action. Moreover, A may be lower or upper triangular (`uplo = 'L'` or `'U'`) and may have a unit or nonunit diagonal (`diag = 'U'` or `'N'`). The four different situations from the standpoint of blocking are summarized in Table 3.

Table 3: _TRSM blocking cases.

Operation	T	Possibilities
$C \leftarrow X,\ TX = C$	lower triangular	(T, `uplo`) = (A, `'L'`) or (A^T,`'U'`)
$C \leftarrow X,\ TX = C$	upper triangular	(T, `uplo`) = (A, `'U'`) or (A^T,`'L'`)
$C \leftarrow X,\ XT = C$	lower triangular	(T, `uplo`) = (A, `'L'`) or (A^T,`'U'`)
$C \leftarrow X,\ XT = C$	upper triangular	(T, `uplo`) = (A, `'U'`) or (A^T,`'L'`)

First, we consider the case $C \leftarrow X,\ TX = C$ where T is lower triangular.



We illustrate by blocking X in three block rows:

$$\begin{aligned}
C_1 &= T_{11}X_1,\\
C_2 &= T_{21}X_1 + T_{22}X_2,\\
C_3 &= T_{31}X_1 + T_{32}X_2 + T_{33}X_3.
\end{aligned}$$

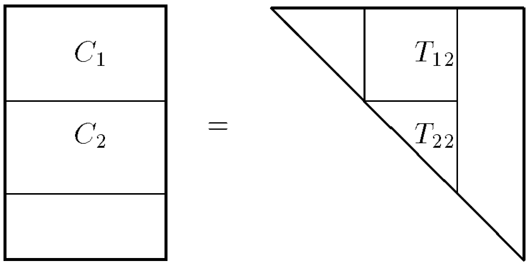
By solving for X in a block forward fashion we can obtain a GEMM-rich procedure:

$$\begin{aligned}
X_1 &\leftarrow T_{11}^{-1}C_1,\\
X_2 &\leftarrow T_{22}^{-1}(C_2 - T_{21}X_1),\\
X_3 &\leftarrow T_{33}^{-1}(C_3 - T_{31}X_1 - T_{32}X_2).
\end{aligned}$$

C_2 and C_3 are updated by calls to _GEMM. The update of C_2 and C_3 with respect to X_1 can be performed by one single call.

Notice, it is only the diagonal blocks T_{ii} that cannot be handled with _GEMM operations. For these we can repeatedly apply the level 2 BLAS operations _TRSV or _GEMV. _TRSV performs matrix-vector products of the form $x \leftarrow T^{-1}x$ where T is a nonsingular triangular matrix described via the variables `uplo`, `diag`, and `trans`. _GEMV performs a matrix-vector multiply and add operation $y \leftarrow \alpha Ax + \beta$, where α and β are scalars, x and y are vectors, and A is a general matrix. Indeed, our model implementation of _TRSM uses one of the two level 2 operations depending on the values of the form parameters. We refer to the next two sections for more information on how the choice between two different level 2 operations are handled.

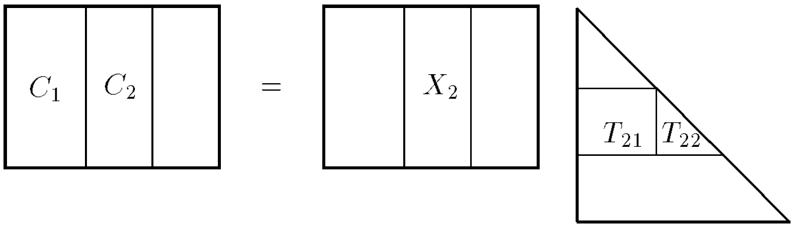
The other three cases listed in Table 3 are similar and require only cursory discussion. For the case $TX = C$ with upper triangular T we have in the i-th step ($i = 2$ and C_1, C_2 are already updated with respect to X_1);



$$\begin{array}{ll} \text{Solve for } X_2: & T_{22}X_2 = C_2, \\ \text{Update } C_1: & C_1 \leftarrow C_1 - T_{12}X_2. \end{array}$$

It follows that if we resolve the block rows of X in reverse order (from bottom to top) we can overwrite C with X.

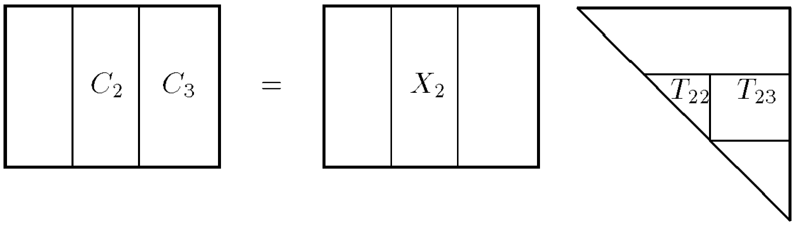
For the case $XT = C$ with T lower triangular we have in the i-th step;



$$\begin{array}{ll} \text{Solve for } X_2: & X_2T_{22} = C_2, \\ \text{Update } C_1: & C_1 \leftarrow C_1 - X_2T_{21}. \end{array}$$

It follows that if we resolve the block columns of X from right to left we can overwrite C with X.

Finally, for the case $XT = C$ with T upper triangular we have in the i-th step;



$$\begin{array}{ll} \text{Solve for } X_2: & X_2T_{22} = C_2, \\ \text{Update } C_3: & C_3 \leftarrow C_3 - X_2T_{23}. \end{array}$$

It follows that if we resolve the block columns of X from left to right we can overwrite C with X.

In the present concept discussion we have omitted several issues that affect the performance of a GEMM-based implementation. Examples include the placement of a possible offset diagonal block, which is determined by the values of `uplo` and `trans`, and the size and orientation of blocks to be referenced and updated. More on this and other design principles are discussed in the next section. The GEMM-based _TRSM algorithm for the model implementation is discussed in more detail in Section 5.5.

4 Design Principles for the Model Implementations

It is possible to implement GEMM-based level 3 routines in several ways. Performance is often significantly affected by different design decisions, even if the underlying BLAS routines (_GEMM and some lower level kernels) are well optimized and show high performance. This section discusses design principles used in the GEMM-based level 3 BLAS model implementations.

4.1 Memory hierarchy model

The GEMM-based level 3 BLAS model implementations are based on a memory hierarchy architecture model with certain characteristics:

- The CPU (or the processor) is connected to the main memory via a cache memory. Data transfers between the CPU and main memory normally go through the cache, where a copy of the data is kept. Accesses to data are handled by a virtual memory system which is responsible for the correctness in handling the data accesses but does not guarantee fast access times.
- The access time to data in the cache memory is constant and independent of where in the cache data reside or in which order data are accessed. For example, accesses to array data in cache are stride independent.
- Updates of data in memory, both reading and writing, are assumed to take more time than just reading data.
- Data transfers between cache and main memory take place in lines of several consecutive array elements, i.e., units of data larger than one element. The elements of a matrix column are assumed to be consecutively stored in memory (as in Fortran). Accordingly, an arbitrary column of a 2-dimensional array requires fewer lines to be transferred than the corresponding matrix row.

There are means to specify different cache characteristics in terms of machine-specific parameters, which are used by the auxiliary routine _CLD (see sections 4.6 and 4.7).

Several of the current high-performance architectures fall within this logical memory hierarchy model. Examples include vector processors with local cache memory and RISC processors with separate on-chip caches for data and instructions. Architectures with multi-level caches can be interpreted within this framework in two different ways: (i) all cache memory is considered as a unit, (ii) only the top level is considered as the cache memory and the lower levels together with the main memory are regarded as one unit of memory. Neither of these interpretations are completely satisfactory. The solution we recommend is to impose that the few underlying routines called by the GEMM-based model implementations utilize the machine characteristics of the target architecture efficiently (including a multi-level cache). Another (and complementary) solution is to apply different levels of blocking with respect to a multi-level memory. However, to guarantee portability as well as high performance we have only implemented blocking with respect to one level of cache memory (see below and Section 8.)

4.2 Blocking strategy

The blocking strategy of the GEMM-based routines should be adapted to the performance characteristics of the underlying BLAS kernels that are called. If the underlying BLAS kernels should have equal and uniform performance for all problems then any GEMM-based implementation would be good enough, but in practice this is not the case.

The performance of the underlying BLAS depend on the size and configuration of the problem, properties of the machine, and how efficient these properties are utilized in the BLAS kernels themselves, i.e., how well they are optimized for the machine. A machine with a vector processor, for example, usually need long vectors or vectors sized to fit in vector registers to perform at its best. Properties of the machine include, for instance, possibilities for pipelining, chaining, use of compound instructions, reuse of data in registers (and in cache for _GEMM), parallelism, etc. For this reason, the GEMM-based routines are designed to utilize the fastest underlying BLAS routines and to supply them with appropriate subproblems.

We see four basic ways to block a GEMM-based level 3 BLAS routine involving a triangular matrix, or a symmetric matrix stored as a triangular matrix. In Figure 1 we illustrate these alternative ways by looking at GEMM-updates $C \leftarrow -CA + C$ within the _TRSM operation. In this case, `side = 'R'`, `uplo = 'U'`, and `trans = 'N'`, i.e., at the outermost level we perform the operation $C \leftarrow CA^{-1}$, where A is triangular and stored in upper triangular format. This operation is covered by the last case illustrated in Section 3.

In each of the four alternatives in Figure 1, there are three GEMM-updates, each illustrated with a different pattern, ▥, ⊠, and ▤ . Notice that some of the blocks involved in the different GEMM-updates overlap and create mixed patterns.

We can identify the four ways by looking at the blocks of the triangular A. In all alternatives, A is partitioned into three uniformly sized triangular diagonal blocks and one smaller triangular diagonal block. This *offset block* may be located in either the lower right corner of the matrix, as in alternatives 1 and 3, or in the upper left corner, as in alternatives 2 and 4. Additionally, three rectangular blocks are needed to cover A.

The subproblems involving triangular diagonal blocks of A have the same dimensions irrespective of in which corner of A the offset block is placed. Therefore, the location of the offset block is not expected to have a significant impact on the per-

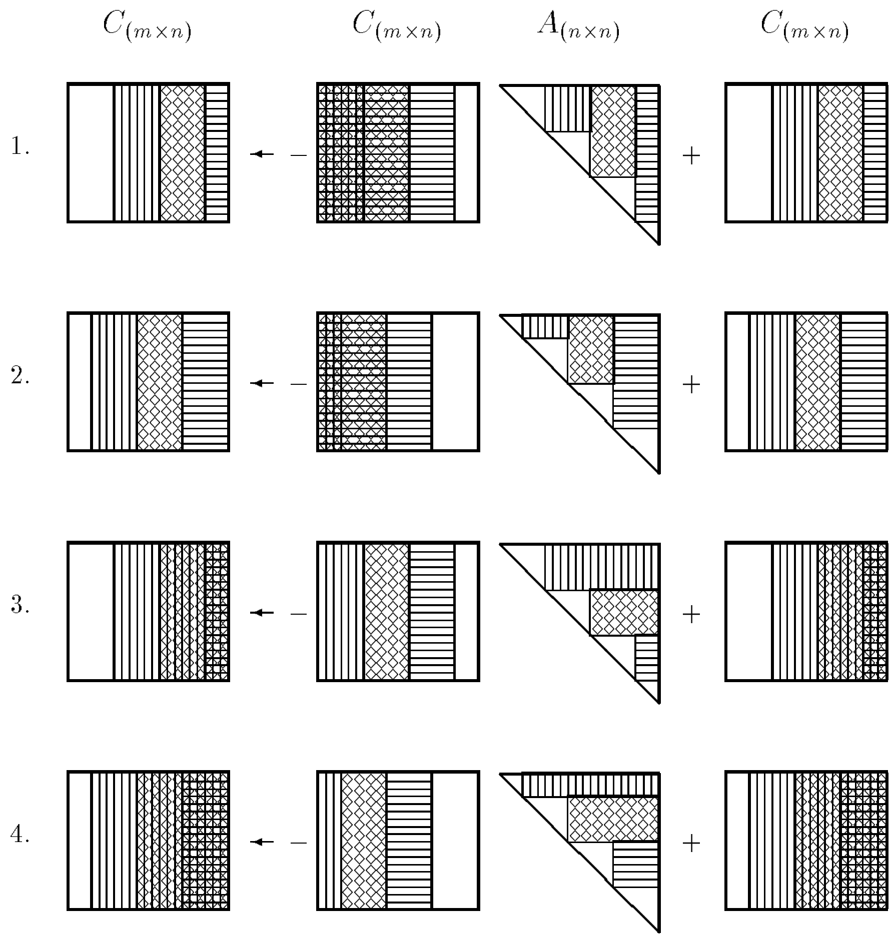


Figure 1: Four basic ways to block _TRSM, `side = 'R'`, `uplo = 'U'` and `trans = 'N'`.

formance of the subproblems involving triangular blocks. However, the subproblems involving rectangular blocks of A have different dimensions in all four alternatives. In alternatives 1 and 2 the rectangular blocks of A have a fixed maximum width, and in alternatives 3 and 4, they have a fixed maximum height. Computations involving rectangular blocks of A can be performed by calls to _GEMM. All four alternatives present a different set of problems for _GEMM. Consequently, the performance of the entire _TRSM operation, is mostly determined by the performance of _GEMM for these sets of subproblems. Which of the four alternatives that gives the best performance may be different for different architecture characteristics. Our approach is to choose a blocking strategy that we expect to be the fastest on most modern memory hierarchy machines. The following guidelines were used to decide which of the four alternatives to select.

- **Update policy**. If data is brought from memory with a high amount of *locality of* *reference*, the data traffic in the memory hierarchy is generally reduced. Locality of reference (or data locality) means that for a sequence of operations, references are kept local to limited portions of consecutive addresses in memory. With a high amount of locality of reference, larger amounts of data can be reused at different levels of the memory hierarchy. In general, we expect that updating, which involves both reading and writing of memory, generates more traffic in the memory hierarchy than just reading data. Since columns of matrices are stored contiguously in Fortran 77, while elements of rows may have a large stride, a higher amount of data locality can, in general, be obtained when accessing block columns of matrices rather than block rows.

  In our implementations we choose to update a block column or a block row of the result matrix completely before the next block column or block row is processed. The blocks are updated consecutively until the whole result matrix is completed. This approach gives a sufficient amount of data locality.

- **Offset block positioning.** The offset block is placed in the corner that causes the adjoining rectangular block to be vertically oriented in memory, or stored as a block column. The rectangular block will often become tall and narrow with fewer and longer columns than rows, enabling a high amount of locality of reference. The columns will always have a minimum height equal to the dimension of the remaining triangular blocks.

  If the matrix has an upper triangular storage format, `uplo = 'U'`, then the offset block is placed in the lower right corner of the matrix and for a lower triangular storage format, `uplo = 'L'`, the offset block is placed in the upper left corner.

In alternatives 1 and 2 of Figure 1 a block column of C is fully updated before the next block of C is processed. In alternatives 1 and 3, the rectangular block adjoining the offset block of A is vertically oriented, i.e., a block column. Only the first blocking strategy fulfills both our criteria and was therefore chosen for our GEMM-based _TRSM implementation.

4.3 Local arrays for consecutive storage of submatrices

An unappropriate size of the leading dimension of a 2-dimensional array may cause particularly heavy traffic in the memory hierarchy when the matrix is referenced repeatedly. Further, only a small fraction of the cache may be utilized. Physical characteristics of the memory hierarchy determines for which leading dimensions these problems will occur.

Local arrays are used extensively in the model implementations to provide properly aligned consecutive storage for temporarily kept blocks of matrices, resulting in improved data locality. The size of the local arrays are determined before compilation. Provided that proper values for the dimensions of the local arrays are specified, these problems with "critical" leading dimensions are effectively avoided. The blocks may be transposed while they are copied to local arrays, in order to possibly fit the succeeding computations in the underlying BLAS routines better. Additionally, symmetric and hermitian blocks of matrices, where only the upper or lower triangular part is stored, can be transformed to general form when they are copied to the local arrays. We get rectangular blocks with long and uniformly sized vectors where the dimensions can be sized to fit, for instance, both vector registers and cache at the same time. Moreover, the general blocks make it possible to enhance the use of _GEMV and _GEMM which are likely to be the fastest level 2 and level 3 routines available on most machines. This generally works well with any type of processor, RISC, CISC, vector processors, etc.

4.4 Underlying BLAS routines

Apart from the level 3 routine _GEMM, the computations are focused on the level 2 routine _GEMV since this is the fastest level 2 routine available, on most machines. The number of different BLAS routines called from the GEMM-based routines are intentionally kept small, in order to reduce the number of necessary machine specific implementations. The underlying BLAS routines are the level 3 routine _GEMM, the level 2 routines _GEMV, _SYR, _TRMV, and _TRSV [7], and the level 1 routines _AXPY, _COPY, and _SCAL [22]. The most significant underlying routines in order for the GEMM-based level 3 BLAS model implementations to achieve high performance are _GEMM and _GEMV.

4.4.1 Level 3 performance obtained with level 2 BLAS

The level 2 BLAS [7] perform matrix-vector operations. For instance, the matrix vector multiply and add operation _GEMV, $y \leftarrow \alpha Ax + \beta y$, where α and β are scalars, x and y are vectors, and A is a general matrix. On a machine with vector registers, or a sufficient number of scalar registers, it is possible to implement level 2 BLAS routines that offer register reuse of a vector. One of the vectors, x or y, is referenced repeatedly and a part of it can be kept in registers between the references. Which vector depends on whether the underlying instructions are arranged to perform _dot or _axpy oriented operations, where _dot denotes the operation $dot \leftarrow dot + x^T y$ and _axpy denotes the operation $y \leftarrow \alpha x + y$.

Cache reuse is usually not associated with the level 2 BLAS since the elements of the matrix A are referenced only once. Cache reuse requires multiple references of a matrix.

If, for instance, the underlying instructions perform dot products, it is possible to keep a large section of the vector x in the cache and reuse it for each row of A. We refer to this approach as vector register reuse using the cache as a large vector register, rather than as cache reuse.

However, if the level 2 routine is called multiple times, it is sometimes possible to attain “true” cache reuse for the level 2 computations. For instance, if _GEMV is called repeatedly with different x and y vectors each time, but with the same A-matrix, it is possible to reuse A in the cache between the calls, provided that A fits properly in the cache. For matrices that do not fit in cache, a blocked approach can be applied. Notice that vector register reuse can still be implemented on top of this. Apart from the overhead caused by multiple calls to _GEMV (parameter checking, etc), this approach makes it possible to reach performance levels usually associated with the level 3 BLAS. The computations of the GEMM-based level 3 BLAS model implementations are structured to utilize this technique extensively in order to attain high and uniform performance.

On machines with the possibility to get explicit control over which data that resides in the cache, the programmer implementing _GEMV may choose to use the cache as a large vector register, for instance, for a large section of the x-vector. This efficiently spoils all chances to reuse the A-matrix and to attain level 3 performance over multiple calls to _GEMV. On machines having a LRU (least recently used) based replacement policy, where the most recently referenced data always resides in the cache, this will not be a problem. Notice that a level 2 implementation that explicitly uses the cache as a large vector register may be faster for one or possibly a few repeated calls. However, for more than just a few calls the approach to reuse a section of x in a vector register and a block of A in the cache, or possibly both x and A in the cache, is likely to be faster.

4.5 Alternative code sections

Alternative code sections performing the same task but calling different underlying BLAS routines are used conditionally to utilize the fastest underlying routine depending on the problem configuration. Mostly, the choice is between a code section that calls the level 2 routine _GEMV and a code section that calls some other level 2 BLAS routine. The alternate code sections have different performance characteristics and the choice between them are controlled by intersection points (*ipx*) in each of the level 3 routines.

Alternative code sections are also used to avoid "critical" leading dimensions and referencing matrices by row.

4.6 Auxiliary routines

The original Fortran 77 model implementations of the level 3 BLAS [9, 10] include two auxiliary subprograms, LSAME and XERBLA. The GEMM-based level 3 BLAS model implementations have two additional auxiliary subprograms, _BIGP and _CLD.

- _BIGP determines which of two alternative code sections, in a GEMM-based level 3 routine, that will be the fastest for a particular problem configuration.

- _CLD determines whether the size of the leading dimension of a 2-dimensional array is appropriate for the target memory hierarchy. A "critical" size of the leading dimension may cause a substantial increase in the amount of data movements in the memory hierarchy, resulting in severe performance degradation. Particularly, this may happen if the array is referenced by row.

For more information about the implementations of _BIGP and _CLD see Section 2 in the companion paper [20].

4.7 Machine-specific parameters

Each of the GEMM-based routines has system dependent parameters which are assigned values at compile time. The parameters specify internal block sizes, cache characteristics, and intersection points for alternate code sections in the GEMM-based routines. Blocking parameters and intersection points will appear in the description of the model implementations (see Section 5). The parameters that specify characteristics of the cache memory are described in [20], where also guidelines for assigning values to the machine-specific parameters are given.

5 High Performance Model Implementations

The model implementations are written in Fortran 77 and are structured to effectively reduce data traffic in a memory hierarchy. A detailed description of the algorithms used in our model implementations for the different level 3 operations is presented in [19]. These descriptions include block partitionings and associated GEMM-based templates for different options of the operations. Since these descriptions are very space-demanding we only give a brief description of the GEMM-based implementations here. This includes the characteristics of each complete GEMM-based level 3 BLAS algorithm summarized in a table that shows the lower level BLAS operations, auxiliary routines, and intrinsic functions used. Moreover, we display local arrays, intersection points (*ipx*) for algorithm variants and blocking parameters associated with the partitionings of the matrices involved. Finally, we discuss the complex case and point to some differences between the real and complex model implementations.

5.1 Symmetric Matrix Multiply

_SYMM performs the matrix multiply and add operation:

- $C \leftarrow \alpha AB + \beta C$, if `side = 'L'`,
- $C \leftarrow \alpha BA + \beta C$, if `side = 'R'`,

where C is a general $m \times n$ matrix, A ($m \times m$ or $n \times n$) is symmetric ($A = A^T$), and stored as an upper or lower triangular matrix.

The implementation consists of four sections of code corresponding to the different values of `side` and `uplo`. Each section consists of an outer sectioning (or blocking) loop to partition the problem into subtasks. In each iteration of this outer sectioning loop three different subtasks are handled that involve:

- a diagonal block of the symmetric matrix A,
- an off-diagonal block of A,
- the transpose of an off-diagonal block of A.

The computations in each of these subtasks are performed by a single call to _GEMM. The blocking strategy for _SYMM is chosen so that a horizontal or vertical block of C

is updated in each iteration of the outer sectioning loop. The block is of size $rc \times n$ (if `side = 'L'`) or $m \times rc$ (if `side = 'R'`).

Since A has triangular storage format some preparations are necessary before _GEMM is invoked in subtasks involving diagonal blocks of A. The triangular diagonal blocks of A can not immediately be processed by _GEMM. Full square diagonal blocks ($rc \times rc$) are created from the non-transpose and the transpose of A using _COPY. The new blocks are stored in a 2-dimensional local array T_1 with general storage format and used (instead of A) in the calls to _GEMM.

T_1, which may be large, is referenced by row when the transpose of A is copied to T_1. Since referencing by row, under certain circumstances may cause increased memory traffic and thereby longer access times, an additional level of blocking is implemented for this suboperation, which reduces the length of the row vectors referenced from rc to c. Vertical $rc \times c$ blocks of the square block T_1 ($rc \times rc$) are referenced as units. The same approach is used to reference the local array in _SYR2K (see Section 5.3).

The local array is only used to change the storage format in this routine. Since all computations are performed by _GEMM we do not need to be concerned with critical leading dimensions and alignment for efficient cache utilization, except for copying the transpose of diagonal blocks of A to T_1 above. We trust that _GEMM handles the memory hierarchy efficiently. However, good performance may well be achieved with local arrays that matches the size of the cache. Notice that no level 2 BLAS is involved in this implementation.

Table 4: Characteristics of the _SYMM implementation.

Level 1 routines called	_COPY
Level 2 routines called	—
Level 3 routines called	_GEMM
Auxiliary routines called	LSAME, XERBLA
Intrinsic functions called	MAX, MIN
Local arrays	T_1 ($rc \times rc$)
Intersection points	—
Blocking parameters	rc, c

The characteristics of the GEMM-based _SYMM is summarized in Table 4.

5.2 Symmetric Rank–k Update

_SYRK performs the rank-k update operation:

- $C \leftarrow \alpha AA^T + \beta C$, if `trans = 'N'`,
- $C \leftarrow \alpha A^T A + \beta C$, if `trans = 'T'`,

where C is $n \times n$, symmetric ($C = C^T$), and stored as an upper or lower triangular matrix. The matrix A is $n \times k$ (if `trans = 'N'`) or $k \times n$ (if `trans = 'T'`).

The implementation consists of four sections of code corresponding to the different values of `uplo` and `trans`. Each of these sections is further divided into two parts which are used conditionally depending on the value of the dimension n. If $n < ip41$, then the first part is used. Otherwise, the second part is used.

The first part uses the level 1 routines _COPY and _SCAL, and the level 2 routine _SYR for computations involving diagonal blocks of C. A square diagonal block of C ($rc \times rc$ and stored in upper or lower triangular format) is copied to the local array T_2 (using _COPY) and if necessary scaled by β (using _SCAL). The block is then rank-1 updated n times (using _SYR) and copied back to C from T_2 (using _COPY) again. If `trans = 'T'` an additional local array T_3 is used for blocks of A^T so that multiple rank-1 updates on diagonal blocks of $C(A^TA)$ can be performed as $T_3T_3^T$ to provide stride one references for _SYR. For small values of k (determined by a second intersection point, $ip42$) the computations are performed directly on the blocks of C, not using T_2 or T_3. Copying blocks to and from T_2 and T_3 imply too much overhead compared to the number of times the blocks are referenced, since k is small. However, if the leading dimension of C is critical then the data traffic in the memory hierarchy may increase substantially when the block of C is referenced. In that case it is important to use T_2, which should be sized to fit safely in the cache. Typically, this is the case if the storage requirements of T_2 is limited to 50–75% of the size of the cache memory.

The second part uses _COPY and _GEMV for computations involving diagonal blocks of C. A rectangular block of A or A^T, depending on the value of `trans` is copied to T_1 ($r \times c$) (using _COPY). Then a square ($r \times r$) diagonal block of C (stored in upper or lower triangular matrix format) is updated by β and $T_1T_1^T$ (using _GEMV). The block in T_1 is then repeatedly replaced by subsequent blocks of A or A^T, and the block of C is repeatedly updated by $T_1T_1^T$ (using _GEMV). A local temporary δ is used to facilitate the scaling of the block of C with β the first time it is referenced. T_1 is reused heavily and should also be sized to fit safely in the cache.

In both parts, the rectangular off-diagonal blocks of C are computed using _GEMM. The sizes of the off-diagonal blocks of C are different in the two parts.

Table 5: Characteristics of the _SYRK implementation.

Level 1 routines called	_COPY, _SCAL
Level 2 routines called	_GEMV, _SYR
Level 3 routines called	_GEMM
Auxiliary routines called	_CLD, _BIGP, LSAME, XERBLA
Intrinsic functions called	MAX, MIN
Local arrays	T_1 ($r \times c$), T_2 and T_3 ($rc \times rc$)
Intersection points	$ip41$ for n, $ip42$ for k
Blocking parameters	r, c, rc

The characteristics of the GEMM-based _SYRK is summarized in Table 5.

5.3 Symmetric Rank–$2k$ Update

_SYR2K performs the rank-$2k$ update operation:

- $C \leftarrow \alpha AB^T + \alpha BA^T + \beta C$, if `trans = 'N'`,
- $C \leftarrow \alpha A^T B + \alpha B^T A + \beta C$, if `trans = 'T'`,

where C is $n \times n$, symmetric ($C = C^T$), and stored as an upper or lower triangular matrix. The matrices A and B are $n \times k$ (if `trans = 'N'`) or $k \times n$ (if `trans = 'T'`).

The implementation consists of four sections of code corresponding to the different values of `uplo` and `trans`. Each section consists of an outer sectioning (or blocking) loop to partition the problem into subtaks. In each iteration of this outer sectioning loop three different subtasks are handled that involve:

- a diagonal block of the symmetric matrix C,
- an off-diagonal block of C,
- the transpose of an off-diagonal block of C.

The computations involving off-diagonal blocks of C are performed by calls to _GEMM. The subtask involving diagonal blocks of C consists of the following three steps:

- Conceptually, one of the operations $T_1 \leftarrow \alpha AB^T$ and $T_1 \leftarrow \alpha A^T B$, depending on the value of `trans`, is performed on rectangular blocks of A and B using _GEMM. T_1 is a local array of size $rc \times rc$.
- The stored upper or lower triangular part (depending on `uplo`) of C_{ii}, a diagonal block of the symmetric C is updated by β times itself (using _SCAL) and by the upper or lower part of T_1 (using _AXPY), so that $C_{ii} \leftarrow T_1 + \beta C_{ii}$.

  Notice that we have to use both _SCAL and _AXPY to perform this fairly simple operation. We lack a level 1 BLAS routine performing the operation $y \leftarrow x + \beta y$ or $y \leftarrow \alpha x + \beta y$.
- C_{ii} is then further updated with the appropriate upper or lower part of the transpose of T_1 (using _AXPY), so that $C_{ii} \leftarrow T_1^T + C_{ii}$.

  Notice that T_1, which may be large, is referenced by row in this operation. As in _SYMM an additional level of blocking is implemented which reduces the length of the row vectors referenced, from rc to c. Vertical blocks ($rc \times c$) of the square block T_1 ($rc \times rc$) are referenced as units.

The result of these three steps is a complete rank-$2k$ update for a diagonal block of C.

Notice that the local array T_1 is needed only to provide a general storage format for the matrix multiply operation. T_1 does not need to fit in the cache. With this approach for _SYR2K all handling of the memory hierarchy becomes local to _GEMM. This approach was first used in [23]. Notice also that no level 2 BLAS is used in this implementation.

The characteristics of the GEMM-based _SYR2K is summarized in Table 6.

Table 6: Characteristics of the _SYR2K implementation.

Level 1 routines called	_SCAL, _AXPY
Level 2 routines called	—
Level 3 routines called	_GEMM
Auxiliary routines called	LSAME, XERBLA
Intrinsic functions called	MAX, MIN
Local arrays	T_1 $(rc \times rc)$
Intersection points	—
Blocking parameters	rc, c

5.4 Triangular Matrix Multiply

_TRMM performs the matrix multiply operation:

- $C \leftarrow \alpha$ op(A) C, if `side = 'L'`,
- $C \leftarrow \alpha$ C op(A), if `side = 'R'`,

where C is an $m \times n$ general matrix, op(A) $(m \times m$ or $n \times n)$ is a unit or non-unit upper or lower triangular matrix, and op$(A) = A$ or A^T.

The implementation consists of eight sections of code corresponding to the different values of `side`, `uplo`, and `trans`. Each of these sections are further divided into two parts which are used conditionally depending on the values of m and n. If `side = 'L'` and $n < ip81$ or if `side = 'R'` and $m < ip83$, then the first part is used. Otherwise, the second part is used.

The first part uses the level 1 routine _COPY and the level 2 routine _TRMV for computations involving triangular diagonal blocks of A. If `side = 'L'` and $n \geq ip82$, then T_3 $(rc \times rc)$ is used to hold the triangular diagonal blocks of A during the _TRMV computations. This guarantees that the blocks of A will reside properly in cache during the calls to _TRMV that follows. The result is efficient cache reuse. If $n < ip82$, the blocks of A are referenced without using T_3 in _TRMV. In this case, using T_3 is considered to represent too much overhead in relation to the number of calls to _TRMV.

If `side = 'R'`, the blocks of C are referenced by row in the calls to _TRMV. This may be very costly if more than one or two calls are made. So, if $m \geq ip83$, the second part is used (see below).

The second part uses _COPY and _GEMV for computations involving triangular diagonal blocks of A. If `side = 'L'`, the following steps are performed repeatedly in a blocked fashion.

- If `trans = 'N'`, the transpose of a triangular diagonal block A_{ii} of A $(c \times c)$ is copied to the local array T_2, using _COPY $(T_2 \leftarrow A_{ii}^T)$.
- The transpose of a rectangular block C_{ij} of C is copied to T_1 $(r \times c)$ using _COPY $(T_1 \leftarrow C_{ij}^T)$.

_GEMV: $T_1 \leftarrow \gamma T_1 T_2 + \delta T_1$

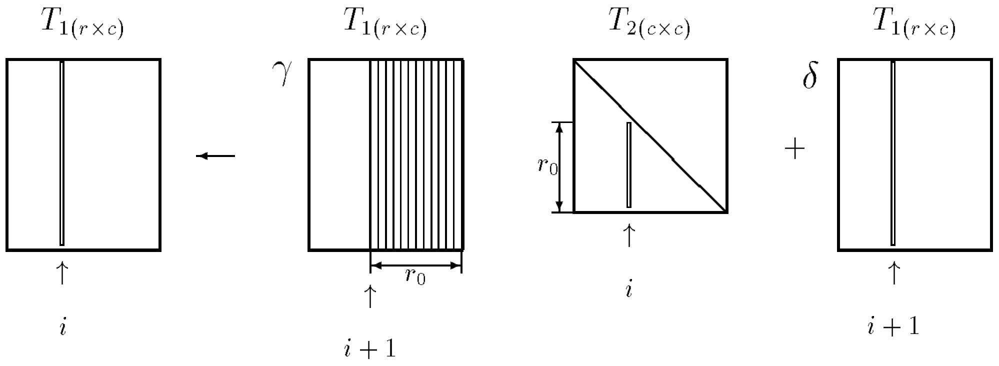


Figure 2: Sample use of _GEMV for triangular diagonal blocks in GEMM-based _TRMM (`side = 'L'`, `uplo = 'U'` and `trans = 'N'`).

- The blocks of C stored in T_1, the blocks of A possibly stored in T_2, and the scalar α are multiplied together using _GEMV, $T_1 \leftarrow \alpha * T_1 * (T_2$ or a block of $A)$. The local array T_1 will be heavily referenced and if properly sized, it will remain in cache between the calls to _GEMV and provide for efficient cache reuse (see Section 4.4). For example, if `side = 'L'`, `trans = 'N'` and `uplo = 'U'`, the operation

  $$T_1(1:ni, i) \leftarrow \gamma T_1(1:ni, i+1:r_0) \cdot T_2(i+1:r_0, i) + \delta T_1(1:ni, i),$$

  where γ is assigned the value α, δ is assigned the value α (if `diag = 'U'`) or $\alpha T_2(i,i)$ (if `diag = 'N'`), is performed with repeated calls to _GEMV.

  In Figure 2 we illustrate one such _GEMV operation. The values for γ and r_0 (the second dimension of the matrix block T_1which varies) are also used to overcome a "deficiency" in _GEMV that appears if the second dimension of the block is zero. Whenever r_0 is zero, it is changed to 1.0 and γ is changed to 0.0, before _GEMV is called. Otherwise, if these changes were omitted, _GEMV would just perform a "quick return" instead of scaling T_1 with δ when $r_0 = 0$. If `trans = 'T'`, the blocks of A are referenced directly without using the local array T_2.

- The result of the multiplication stored in T_1 is copied back to the matrix C using _COPY $(C_{ij} \leftarrow T_1^T)$.

A similar approach is used if `side = 'R'`. The differences are (i) the blocks of C are not transposed when they are copied to the local array T_1, (ii) the results are stored back directly to C instead of via T_1. The diagonal blocks of A are copied to T_2 and transposed when `trans = 'T'` instead of when `trans = 'N'`.

In both parts, the computations involving rectangular off-diagonal blocks of A are performed using _GEMM. In the first part, _GEMM performs the scaling of C ($C \leftarrow \alpha C$) before the computation involving a triangular block starts, and the operation

$C \leftarrow \alpha\text{op}(A)C{+}C$ or $C \leftarrow \alpha C\text{op}(A){+}C$, is performed after completion of the triangular computation. In the second part the scaling of C is performed in the calls to _GEMV. Notice that the sizes of the blocks involved are different between the two parts.

If `side = 'L'`, a horizontal block of C is updated in each iteration of the outermost loop. Otherwise, if `side = 'R'`, a vertical block of C is updated in each iteration.

Table 7: Characteristics of the _TRMM implementation.

Level 1 routines called	_COPY
Level 2 routines called	_GEMV, _TRMV
Level 3 routines called	_GEMM
Auxiliary routines called	_CLD, _BIGP, LSAME, XERBLA
Intrinsic functions called	MAX, MIN, MOD
Local arrays	T_1 $(r \times c)$, T_2 $(c \times c)$, T_3 $(rc \times rc)$
Intersection points	$ip81$ and $ip82$ for n, $ip83$ for m
Blocking parameters	r, c, rc

The characteristics of the GEMM-based _TRMM is summarized in Table 7.

5.5 Triangular System Solve

_TRSM solves for X in the matrix equation:

- $\text{op}(A)\ X = \alpha C$, if `side = 'L'`,
- $X\ \text{op}(A) = \alpha C$, if `side = 'R'`,

where C is an $m \times n$ general matrix, $\text{op}(A)$ $(m \times m$ or $n \times n)$ is a unit or non-unit upper or lower triangular matrix, and $\text{op}(A) = A$ or A^T. The solution X overwrites C $(C \leftarrow X)$.

The implementation is very similar to _TRMM and consists of eight sections of code corresponding to the different values of `side`, `uplo`, and `trans`. The differences in these code sections compared to _TRMM are:

- The direction of some of the blocking loops are reversed, so that the blocks and columns of blocks of C are read (operand) and updated (result) in the reversed order compared to _TRMM.
- Only one call to _GEMM is needed in each part of the eight code sections. This call is placed before the computations involving triangular blocks of A and performs both the scalar update of blocks of C and the blocked matrix multiply. Conceptually, the operation is $C \leftarrow -\text{op}(A)C + \alpha C$ or $C \leftarrow -C\text{op}(A) + \alpha C$.
- The values assigned to the variables γ and δ in _TRSM are different from those assigned in _TRMM. Scalar division is involved if `diag = 'N'` (for dividing by the diagonal elements of A).

- If side = 'R', T_1 also keeps the result vectors from _GEMV until the block is complete. Then T_1 is copied back to C. This is necessary since, in _TRSM, blocks of C are also read after they have been updated. The operand C (T_1) needs to be the same as the result block C (T_1) in calls to _GEMV. This is not the case in _TRMM where C is not read after being updated.
- The intersection points $ip81$, $ip82$, and $ip83$ in _TRMM correspond to $ip91$, $ip92$, and $ip93$, respectively, in _TRSM.

Table 8: Characteristics of the _TRSM implementation.

Level 1 routines called	_COPY
Level 2 routines called	_GEMV, _TRSV
Level 3 routines called	_GEMM
Auxiliary routines called	_CLD, _BIGP, LSAME, XERBLA
Intrinsic functions called	MAX, MIN, MOD
Local arrays	T_1 $(r \times c)$, T_2 $(c \times c)$, T_3 $(rc \times rc)$
Intersection points	$ip91$ and $ip92$ for n, $ip93$ for m
Blocking parameters	r, c, rc

The characteristics of the GEMM-based _TRSM is summarized in Table 8.

5.6 The Complex Case

The definitions of level 3 BLAS for single complex and double complex data differ from the real counterparts in some aspects. First of all, we have the additional routines for hermitian matrices, _HEMM, _HERK, and _HER2K. Apart from the complex routine _SYMM which is a direct translation of the real _SYMM, the remaining routines also have other differences than just the type. The symmetric rank updates _SYRK and _SYR2K do not allow, `trans = 'C'`, in the complex versions as opposed to the real versions where `'C'` is interpreted as `'T'`. In the complex _TRMM and _TRSM, `trans = 'C'`, means the conjugated transpose whereas in the real cases `'C'` is treated like `'T'`. The conjugated transpose cases obviously bring additional code into the complex versions of _TRMM and _TRSM.

For the GEMM-based level 3 BLAS the differences between the real and the complex routines are greater than in the original Fortran 77 model implementations [10]. This is, to a great extent, due to the use of underlying level 1 and level 2 BLAS routines. These routines have differences between the real and complex versions, which need to be handled by the GEMM-based routines. For example, the real _SYRK calls the underlying level 2 symmetric rank-1 update _SYR, which has no complex counterpart. In the complex _SYRK we use the complex level 1 routine _AXPY to perform symmetric rank-1 updates. Moreover, we have extended the use of local arrays to store the conjugated transpose of subarrays and in general to facilitate the preparation of subproblems to fit the underlying routines well. Code sections that copy the conjugated transpose of blocks to local arrays are implemented with inline code using the intrinsic functions CONJG/DCONJG, since no convenient BLAS routine exists. The hermitian routines _HEMM, _HERK, and _HER2K are developed from their symmetric counterparts _SYMM, _SYRK, and _SYR2K. They contain inline code and use local arrays more frequently. The intrinsic routines REAL/DBLE, CMPLX/DCMPLX, and CONJG/DCONJG, are used for appropriate data conversion. We assume the same definitions as in the original BLAS and LAPACK for all intrinsic functions. Notice that the scalars α and β in _HERK and β in _HER2K are defined to be real and not complex.

6 GEMM–Based Level 3 BLAS Benchmark

Many people have put a lot of effort into developing fast level 3 BLAS since the specification was published in 1990 [9, 10]. Some vendors provide highly optimized BLAS for their machines, see for example [2, 1, 16, 4, 24], while others provide optimized versions of some or none of the routines. Vendor-independent groups have also developed tuned level 3 kernels for different machines, for example [23, 17, 13, 6, 14], where some are based on the GEMM-based concept [17, 6, 14].

Today different implementations with different performance characteristics coexist and it is becoming more important to evaluate different implementations thoroughly.

The GEMM-based benchmark measures the performance of an arbitrary set of level 3 BLAS implementations, specified by the user, and compares it with the performance of the GEMM-based level 3 BLAS model implementations, permanently included in the benchmark. The level 3 BLAS implementations specified by the user are linked with the benchmark program. When the benchmark executes, timings are performed according to specifications given in an input file. The user may design her/his own tests or use the enclosed input files (see [20]). The following output results are eligible for presentation:

- A collected "mean value" statistic, calculated from the performance results of the user-specified level 3 BLAS routines for the problem configurations specified in the input file.

- Tables, that show measured performance results in Mflops for each routine and choice of parameters. Both the user-specified and the GEMM-based level 3 BLAS routines are timed and their performances are compared.

The tables are intended for program developers and others who require detailed performance information. It is possible to choose which of the following results that are to be presented:

- Performance of the GEMM-based level 3 BLAS routines in Mflops.

- Performance of the user-specified level 3 BLAS routines in Mflops.

- Performance of the user-specified _GEMM routine in Mflops.

- GEMM-efficiency of the user-specified level 3 BLAS routines.

- GEMM-ratio.

Results are presented for each routine and problem configuration specified in the input file. Each item in the listing above corresponds to a column of the output tables (see [20] for more information). The last two items are defined as follows. The GEMM-efficiency is intended to illustrate how close to the "practical" peak performance a routine reaches, which is defined as the performance of the highly optimized _GEMM routine specified by the user for the given problem configuration. Notice that the "practical" peak performance can be considerably lower than the maximal theoretical performance of the architecture considered. The performance of the user-specified level 3 BLAS routine is compared with the performance of the user-specified _GEMM routine. Let $Perf(x)$ denote the the performance in Mflops of x. Then

$$\text{GEMM-efficiency} = \frac{Perf(\text{user-specified level 3 routine})}{Perf(\text{user-specified \_GEMM routine})}.$$

The GEMM-efficiency is measured using a choice of parameters for _GEMM which, in this respect, "corresponds" to the problem configuration for the level 3 routine it is compared with.

The GEMM-ratio is the performance of a GEMM-based level 3 BLAS routine compared with the user-specified implementation of the same routine, i.e.,

$$\text{GEMM-ratio} = \frac{Perf(\text{GEMM-based routine})}{Perf(\text{user-specified level 3 routine})}.$$

For a vendor-supplied level 3 BLAS library we would expect to have all GEMM-ratios less than 1. However, this is not always the case (e.g., see results in [17, 18] and Section 7). A value greater than one implies that the GEMM-based implementation is faster than the user-specified implementation for the given problem configuration.

The collected "mean value" statistic provides a comprehensive performance result of the user-specified routines that is easy to compare with other level 3 BLAS implementations, and between different machines. This result consists of a tuple (x, y), where x is the mean value of the GEMM-efficiency and y is the mean value of the performance of _GEMM. The GEMM-efficiency is measured for the routines and choice of parameters that are specified in the input file and the performance of _GEMM is measured for corresponding problem configurations. Provided that _GEMM is well implemented, y represents the average "practical" peak performance of the target computer system, for the specified problem configurations. The average performance of the remaning user-specified implementations can be approximated as $x \cdot 100$ percent of the average "practical" peak performance y.

To further standardize the result we have included a pair of "canonical" input files called _MARK01 and _MARK02, which are explained in [20]. The user can also use "customized" input files to obtain results for problem configurations of interest.

7 Performance and Benchmark Results

We have tested our GEMM-based model implementations and performance evaluation benchmark on several platforms (vector as well as RISC-based), including Alliant FX/2800, IBM 3090 VF, IBM RS6000, IBM SP2, Intel PARAGON, NEC SX-3, Parsytec GC/PowerPlus and Silicon Graphics (SGI) Indy. In this section we report some of these results, focusing on modern high-performance architectures. Sample performance results for the vector machine IBM 3090 VF can be found in [17, 18]. The correctness of the GEMM-based model implementations were verified by the testing program accompanying the original model implementations. The measured performance results presented here are all for double precision real data (64 bits floating point numbers) and were obtained by using the GEMM-based performance evaluation benchmark. The exception is the results for NEC SX-3 for which single precision real data corresponds to 64 bits floating point arithmetic. Corresponding results for double precision complex data are very similar.

7.1 Performance results of the GEMM-based model implementations

In the first set of tables we compare the performance of the GEMM-based routines with optimized vendor-supplied level 3 BLAS.

Tables 9 and 10 show GEMM-ratios for level 3 BLAS provided in the IBM ESSL library [16]. The results are obtained on IBM RS6000 250 and IBM RS6000 530H (Table 9) and on a thin and wide node, respectively, of the scalable IBM SP2 system (Table 10). The underlying routines of the GEMM-based library are from ESSL, except for the results on IBM RS6000, where we used our own developed Fortran 77 implementation of DGEMV (denoted POL-DGEMV in the tables). We decided to implement POL-DGEMV when we discovered that ESSL DGEMV did not perform satisfactory when the matrix already redsides in cache. We could not get level 3 performance from the computations as described in Section 4.4.1. The problem was an eight cycle halt occurring in ESSL DGEMV whenever elements from a new line of data are loaded [2]. POL-DGEMV is implemented with unrolling and data fetching in advance similar to algorithmic prefetching as described in [2].

The corresponding results for the Paragon Basic Math Library (`lkmath`) on a single node of the Intel PARAGON are displayed in Table 11. The underlying routines of the GEMM-based library are from `lkmath`, except for DGEMV for which we use an optimized assembler version (denoted KD-DGEMV in the tables) [5]. This routine stores parts of A in cache memory and thereby makes it possible to attain level 3 performance of consecutive _GEMV operations where the A-block is kept fix but x is varied (see Section 4.4.1).

Table 12 shows similar results for the vendor-supplied level 3 BLAS (`libblas`) on SGI Indy equipped with MIPS R4000 and R4400 processors. The underlying routines of the GEMM-based library used come from the SGI library `libblas`.

The level 3 libraries provided by the vendors are considered to be highly optimized and perform well on the respective target architectures. Our results clearly demonstrate that the portable GEMM-based model implementations are competetive with all three and that they all can be improved further. When level 2 BLAS operations are imperative as in DSYRK, DTRMM and DTRSM, the corresponding GEMM-based routines do not fully match the best results. However, if the portability claim is dropped, inline source code implementing consecutive level 2 operations adapted to a specific target machine can resolve this problem (see Section 8). On the other hand, when no level 2 BLAS operations are required as in DSYMM and DSYR2K even the best-performed vendor-manufactured routines can gain from the GEMM-concept.

Extensions and modifications of high-performance architectures are introduced regularly. Now and then, we also see new architectures appear on the market. These circumstances show the benefits of our GEMM-approach, since we only require a few routines to be optimized for the target architecture (see Section 4.4). One example of a recent scalable high-performace architecture is the Parsytec GC/PowerPlus, based on the PowerPC microprocessor as computing nodes. Presently, Parsytec does not offer an optimized level 3 BLAS for this architecture. Fortunately, there exists an optimized DGEMM routine for the machine (developed by Bernhard Przywara and denoted BP-DGEMM in the tables) which enabled us to try out the GEMM-based model implementations. BP-DGEMM builds on the work in [8]. The remaining underlying routines of the GEMM-based library are from the original level 1 and 2 BLAS model implementations. In Table 13 we show the GEMM-ratios from a single node of Parsytec GC/PowerPlus. The comparison is here with the original level 3 BLAS model implementations from netlib. In the right-most part of Table 13 we show results where the original DGEMV has been replaced by POL-DGEMV. These clearly demonstrates the benefits of using an optimized level 2 _GEMV routine as well.

One way to invoke parallelism in the GEMM-based level 3 BLAS is to use parallel versions of the underlying BLAS kernels. At minimum this implicit approach requires a well-optimized parallel version of _GEMM. In Table 14 we display multiprocessor performance results of the GEMM-based DSYR2K executing on an ALLIANT FX/2816 (one-processor peak performance in double precision real arithmetic is 40 Mflops), only using a parallel DGEMM. We show the performance in Mflops, the parallel speedup S_p on p processors, and the parallel efficiency E_p for DSYR2K. Moreover, we show the corresponding Mflops results for the parallel DGEMM routine from the ALLIANT library `libalgebra`, and the GEMM-efficiency defined as the ratio between the performance in Mflops of DSYR2K and DGEMM for the same problem sizes. A number close to 1 indicates that DSYR2K performs as well as the parallel DGEMM. Our results demonstrate that the GEMM-based approach can also be appropriate for parallel processing (especially in a shared memory environment). Notably, the parallel DGEMM in `libalgebra` and the GEMM-based level 3 BLAS make it possible to create a multiprocessor version of LAPACK on ALLIANT FX/2800 systems.

Finally, we are able to show some results of NEC SX-3, a top of the line high-performance computer. In Tables 15 and 16 we display one-processor performance results from NEC SX-3 Model 22. The theoretical peak performance of a single processor is 2.75 Gflops. According to NEC System Laboratory, the performance results from initial testings of the GEMM-based model implementations on NEC SX-3 are very impressive, and they are currently implementing them on both NEC SX-3 and NEC SX-4.

7.2 Benchmark results

We are collecting results from the GEMM-based level 3 BLAS performance evaluation benchmark. In tables 17 and 18 we present some benchmark results for the architectures considered in Section 7.1, using the canonical input files DMARK01 and DMARK02. Columns 2–4 of the tables display the GEMM-efficiency of the GEMM-based model implementations, the original Fortran model implementations [9, 10] and the vendor-supplied level 3 BLAS, respectively. Column 5 shows the mean value of the performance (measured in Mflops) of the user-specified DGEMM. For IBM RS6000 we provide two GEMM-efficiency numbers for the GEMM-based model implementations. One where all underlying routines are from the ESSL library, and one where all routines except DGEMV (for which we use POL-DGEMV) are from ESSL. As before, all underlying routines for IBM SP2 are from ESSL. We also provide two numbers for the GEMM-based model implementations on Intel PARAGON. One where all underlying routines are from the `lkmath` library. In the other case we replaced DGEMV in `lkmath` with KD-DGEMV. Presently, Parsytec does not provide an optimized level 3 BLAS library. For the benchmark results of the GEMM-based model implementations we use POL-DGEMV and BP-DGEMM. For SGI Indy all underlying routines are from the library provided by the vendor.

The GEMM-efficiency is an overall performance number that measures how much of the computations are performed in terms of GEMM operations. A number close to one means that the level 3 computations (specified by the user in an input file) are executed with almost the same performance as the user-specified _GEMM routine for similar problem configurations. Notice that it is not always possible to obtain a GEMM-efficiency equal to one, since some operations by default cannot be expressed in terms of GEMM. From the benchmark results in tables 17 and 18 we see that the GEMM-based model implementations show a factor 2-3 times better performance (measured in terms of the GEMM-efficiency) than the original Fortran 77 model implementations [9, 10]. When collecting the results for the original Fortran 77 model implementations we imposed all kinds of optimizations that were provided by the Fortran compilers. We selected the ones that gave the best performance. Without these compiler optimizations, the differences between the two Fortran 77 libraries would have been even larger. In this context, we also mention that we do not recommend any "fancy" optimization flags to be set when compiling the GEMM-based model implementations. (More on this in Section 3.4 of the companion paper [20].) Moreover, we see that GEMM-based model implementations also compete well with the vendor-manufactured level 3 BLAS libraries, and show in some cases even better results.

8 Additional Techniques for Effective Use of a Memory Hierarchy

With the model implementations we provide a high performance GEMM-based level 3 BLAS library which is portable between different memory hierarchy machines. If we focus on a specific target architecture it might be possible to gain performance even after tuning the machine specific parameters of the model implementations. Here we discuss two common techniques that are useful for targeted optimization.

Our motto is that the machine characteristics of a target architecture should as much as possible be hidden and utilized in the underlying BLAS. Therefore, we first of all recommend further optimization efforts to be focused on _GEMM and the underlying level 1 and 2 BLAS routines. The optimization techniques described in this section and Section 4 are likely to be applicable to the underlying routines.

8.1 A second level of blocking

The model implementations have no specific blocking for a possible translation look-aside buffer (TLB) or a second level cache. A TLB holds real addresses to a certain number of pages in the main memory. Usually the most recently touched pages. References to pages in main memory whose addresses are missing in the TLB need to be translated from virtual addresses to real addresses which may take considerable time. An additional level of blocking for a second level of cache memory or to make references local to the pages currently pointed to by the TLB, could possibly increase the performance. TLB blocking is, for instance, implemented in the IBM ESSL library [1]. However, if TLB blocking is already implemented in the underlying _GEMM routine, then to implement TLB blocking in the GEMM-based routines would only cause marginal performance improvements.

8.2 Inlining source code and register blocking

Even if the underlying routines are highly optimized and we get level 3 performance for some of the level 2 computations (see Section 4.4.1), it is sometimes possible to achieve even higher performance with well-tuned inline code than with multiple calls to level 2 BLAS. This is due to limitations in the BLAS operations themselves. Inline code makes it possible to improve the ratio between the number of computations and the number of loads and stores. This is illustrated by the following example.

Assume that we wish to multiply a block A_{ij} of a matrix A with two vectors, x and y, which all reside in cache. We may accomplish the operation by calling _GEMV twice to perform two matrix-vector multiplications, $A_{ij}x$ and $A_{ij}y$. On machines where the arithmetic instructions operate solely on data in registers, the elements of A_{ij} are necessarily loaded into registers twice, and the elements of x and y once for each vector. If we instead perform a matrix-matrix multiplication, $A_{ij}[x,y]$, where the vectors are treated as a matrix with two columns and if there are sufficient number of registers available, it is only necessary to load the elements of A_{ij} into registers once. The elements of x and y still only need to be loaded once. The matrix multiply operation could be accomplished by a single call to _GEMM or by using efficient inline code. Now, assume that there are more than two vectors involved and that the vectors have different lengths. Perhaps they constitute a triangular block (as in _TRMM and _TRSM). In this case, using _GEMM would be a poor solution since _GEMM only operates on rectangular blocks. Therefore, inline code is appropriate in order to avoid loading the elements of A twice.

Since the number of registers usually are not sufficient to keep the entire operands, it is necessary to use some sort of register blocking to minimize the number of loads and stores and attain efficient register reuse. One way to implement register blocking is similar to cache blocking but in this case the blocks are small enough to fit in the registers. At the source code level this may be implemented with a technique called loop unrolling [1]. The technique unwinds some of the iterations in a loop and perform them in a single iteration. Loop unrolling can often be applied to several nested loops and the technique can operate on different matrix dimensions simultaneously. The statements inside the innermost loop form a tiny blocked subproblem, where the reusable data fit in the registers and the amount of non-reusable data is minimized. Additionally, there must be some "clean-up" code for the remaining iterations that are not handled by the unrolled code.

For the GEMM-based routines, it may be possible to gain some performance by replacing calls to the level 1 and 2 BLAS with proper inline source code. However, it would not be possible to maintain high performance across a wide range of processors. Vector and RISC processors, for example, require quite different source code to perform well. Moreover, the routines would become much more dependent on different compilers. Even for the limited class of RISC-based machines, it is not sufficient with a single unrolled Fortran code. Our tests show that the GEMM-based routines often reach better performance with calls to vendor-supplied level 2 routines than with common portable inline code. One explanation can be found in vendor-supplied level 2 routines that also use other more or less machine- and/or compiler-specific techniques to gain performance. For example, the IBM ESSL library uses a technique called algorithmic prefetching [2], in order to continuously feed the processor with useful data and avoid delays caused by memory accesses.

9 Conclusions

The objective of the GEMM-based approach is to express the structured matrix multiplications problems handled by the level 3 BLAS (including triangular solve with multiple right hand sides) in terms of general matrix multiply and add (GEMM) operations and a small amount of level 2 and level 1 operations. Since the GEMM operation delivers the best performance (measured in Mflops) of all level 1, level 2 and level 3 BLAS operations, the goal is to perform as much as possible of the computations in terms of _GEMM. This is effected by appropriate partitionings of the matrices involved in the level 3 operation. If the underlying routines, i.e., _GEMM and some level 1 and level 2 BLAS kernels are efficiently optimized for the target machine, the GEMM-based level 3 BLAS model implementations provide:

- Efficient use of vector instructions (compound instructions, chaining, etc.), through _GEMM, level 1 and level 2 BLAS routines.
- Register and vector register reuse, through _GEMM and level 2 BLAS routines.
- Efficient cache reuse, through internal blocking, use of local arrays, and through _GEMM.

- Column-wise referencing, for problems that would cause severe performance degradation with row-wise referencing (except for reference patterns in underlying BLAS routines).

- Parallelism, through automatic parallelization by a compiler, or by using parallel underlying BLAS kernels.

- A level 3 BLAS library based on unconventional underlying matrix multiply algorithms like, for example, Strassen's or Winograd's algorithms (e.g., see [25, 26, 15, 11]).

We have also contributed with the GEMM-based level 3 BLAS performance evaluation benchmark. This program package facilitates the evaluation and comparison between different level 3 BLAS libraries. The benchmark compares a user-specified level 3 BLAS library (e.g., a vendor-supplied library) with the GEMM-based model implementations. Besides performance results (measured in Mflops) the benchmark evaluates the user-specified library in terms of GEMM-ratios and the GEMM-efficiency. The GEMM-efficiency measures how much of the computations are performed in terms of GEMM operations. A number close to one means that almost all computations are GEMM operations. The GEMM-ratio measures the relative performance of a user-specified routine with resepct to the corresponding GEMM-based routine.

Performance results from extensive testings show that the GEMM-based model implementations in Fortran 77 is a high performance level 3 BLAS library that is portable over a wide spectrum of different memory hierarchy architectures. The model implementatons are competetive with vendor-manufactured level 3 BLAS libraries, and in some cases even better. Perhaps, more importantly, the GEMM-based approach only requires a few underlying routines to be optimized when new or extensions and modifications of high-performance architectures are introduced on the market.

To conclude, our GEMM-based concept is the correct level of abstraction for developing a level 3 BLAS library which is both portable over a spectrum of memory hierarchy arctitectures and can deliver near to practical peak performance.

Acknowledgements

We are grateful to Ramesh Agarwal and Fred Gustavson, IBM Thomas J. Watson Research Center, Yorktown Heights, for fruitful discussions relating to this work. We are also grateful to Bernhard Przywara, IWRF, University of Heidelberg for providing us with an optimized DGEMM routine for the Parsytec GC/PowerPlus, and Naoki Iwata, NEC Systems Laboratory Inc. for installing the model implementations and running the benchmark on the NEC SX-3.

The other performance results reported have been performed at the Parallel Computing Laboratory, Bergen University, Norway (Intel Paragon), PDC, Royal Institute of Technology (IBM SP2), NSC, Linköping University (Parsytec GC/PowerPlus), Umeå University (Alliant FX/2800, IBM RS6000, Silicon Graphics Indy) and Supercomputer Center North (IBM 3090VF).

Financial support has been received from the Swedish National Board of Industrial and Technical Development under grant NUTEK 89-02578P.

References

[1] R.C. Agarwal, F.G. Gustavson, and Z. Zubair. Exploting functional parallelism of POWER2 to design high-performance numerical algorithms. *IBM J. Res. Develop.*, 38(5):563–576, September 1994.

[2] R.C. Agarwal, F.G. Gustavson, and Z. Zubair. Improving performance of linear algebra algorithms for dense matrices using algorithmic prefetching. *IBM J. Res. Develop.*, 38(3):265–275, May 1994.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenny, S. Ostrouchov, and D. Sorensen. *LAPACK Users Guide*. SIAM Publications, 1992. ISBN 0–89871–294–7.

[4] Intel Corporation. Paragon Basic Math Library Performance Report. Technical Report 312936-001, Intel Supercomputer Division, Beaverton, Oregon, October 1993.

[5] K. Dackland. Design Issues and the Performance of Level 1 and Level 2 Kernels on Intel i860-based Platforms. Report UMINF-95.xx, Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden, 1995. *To appear.*

[6] M. J. Dayde, I. S. Duff, and A. Petitet. "A Parallel Block Implementation of Level-3 BLAS for MIMD Vector Processors". *ACM Trans. Math. Softw.*, 20(2):178–193, June 1994.

[7] J. Dongarra, J. Du Croz, S. Hammarling, and R. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 14:1–17, 18–32, 1988.

[8] J. Dongarra, P. Mayes, and G. Radicati di Brozolo. The IBM RISC System 6000 and Linear Algebra Operations. *Supercomputer*, 8(4):15–30, 1991.

[9] J. J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Software*, 16(1):1–17, March 1990.

[10] J. J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. Algorithm 679: A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs. *ACM Trans. Math. Software*, 16(1):18–28, March 1990.

[11] C.C. Douglas, M. Heroux, G. Slishman, and R.M. Smith. GEMMV: A Portable Level 3 BLAS Winograd Variant of Strassen's Matrix-Matrix Multiply Algorithm. *J. Comp. Physics*, 110:1–10, 1994.

[12] K. Gallivan, W. Jalby, U. Meier, and A.H. Sameh. Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. *Int. J. Supercomput. Appl.*, 2:12–48, 1988.

[13] H. Grasemann. Optimization of Level 3 BLAS for SIEMENS VP Systems. Tech. rep. No. 38.89, University of Karlsruhe, Computer Center, September 1989.

[14] M. Green. High Performance Level 3 BLAS. A KSR Implementation. Working note, Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK, April 1994.

[15] N.J. Higham. Exploiting Fast Matrix Multiplication Within the Level 3 BLAS. *ACM Trans. Math. Software*, 16(4):352–368, 1990.

[16] IBM. *Engineering and Scientific Subroutine Library, Guide and Reference*, January 1994. SC23–0526–01.

[17] B. Kågström, P. Ling, and C. Van Loan. High Performance GEMM-Based Level 3 BLAS: Sample Routines for Double Precision Real Data. In M. Durand and F. El Dabaghi, editors, *High Performance Computing II*, pages 269–281, Amsterdam, 1991. North–Holland.

[18] B. Kågström, P. Ling, and C. Van Loan. Portable High Performance GEMM–Based Level 3 BLAS. In R.F. Sincovec et al., editors, *Parallel Processing for Scientific Computing*, pages 339–346, Philadelphia, 1993. SIAM Publications.

[19] B. Kågström, P. Ling, and C. Van Loan. GEMM-Based Level 3 BLAS: Algorithms for the Model Implementations. Report UMINF-94.13, Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden, December 1994.

[20] B. Kågström, P. Ling, and C. Van Loan. ALGORITHM XYZ. GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. Report UMINF-95.19, Department of Computing Science, Umeå University, S-901 87 Umeå, Sweden, 1995. *Submitted to ACM Trans. Math. Software.*

[21] B. Kågström and C. Van Loan. GEMM-Based Level 3 BLAS. Technical Report CTC91TR47, Department of Computer Science, Cornell University, December 1989.

[22] C. Lawson, R. Hanson, R. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Software*, 5:308–323, 1979.

[23] P. Ling. A Set of High Performance Level-3 BLAS Structured and Tuned for the IBM 3090 VF and Implemented in Fortran 77. *The Journal of Supercomputing*, 7(3):323–355, September 1993.

[24] Q. Sheik, V. Phuong, Y. Chao, and M. Merchant. Implementation of the Level 2 and 3 BLAS on the CRAY Y-MP and the CRAY-2. *The Journal of Supercomputing*, 5(4):291–305, February 1992.

[25] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[26] S. Winograd. Some Remarks on Fast Multiplication of Polynomials. In J. F. Traub, editor, *Complexity of Sequential and Parallel Numerical Algorithms*, page 181, New York, 1973. Academic Press.

Table 9: GEMM-ratios for the IBM ESSL library on IBM RS/6000 250 and IBM RS/6000 530H. Underlying routines: ESSL and POL-DGEMV.

Routine	Dimensions		IBM RS/6000 250				IBM RS/6000 530H			
DSYMM	M	N	SIDE, UPLO				SIDE, UPLO			
			'L','U'	'L','L'	'R','U'	'R','L'	'L','U'	'L','L'	'R','U'	'R','L'
	32	256	1.00	1.00	0.98	0.97	1.00	1.00	1.01	1.01
	64	256	0.99	0.99	0.98	0.98	1.00	1.00	1.00	1.00
	96	256	0.96	0.95	0.98	0.98	1.09	1.08	0.99	0.99
	256	32	0.90	0.89	1.00	1.00	0.97	0.96	1.00	1.00
	256	64	0.94	0.94	1.01	1.00	0.99	0.99	1.00	1.00
	256	96	0.96	0.96	1.02	1.05	1.00	1.00	1.00	1.00
	256	256	0.98	0.98	1.02	1.02	1.01	1.01	0.99	0.99
DSYRK	N	K	UPLO, TRANS				UPLO, TRANS			
			'U','N'	'U','T'	'L','N'	'L','T'	'U','N'	'U','T'	'L','N'	'L','T'
	32	256	0.74	0.78	0.74	0.79	0.58	0.55	0.58	0.54
	64	256	0.64	0.71	0.64	0.71	0.64	0.65	0.63	0.64
	96	256	0.73	0.81	0.72	0.81	0.68	0.72	0.67	0.71
	256	32	0.80	0.83	0.80	0.83	0.76	0.79	0.76	0.78
	256	64	0.83	0.82	0.82	0.81	0.81	0.84	0.82	0.84
	256	96	0.80	0.87	0.79	0.86	0.79	0.82	0.78	0.82
	256	256	0.79	0.85	0.79	0.85	0.81	0.84	0.80	0.84
DSYR2K	N	K	UPLO, TRANS				UPLO, TRANS			
			'U','N'	'U','T'	'L','N'	'L','T'	'U','N'	'U','T'	'L','N'	'L','T'
	32	256	0.98	0.98	0.97	0.98	0.96	0.96	0.96	0.96
	64	256	1.06	1.08	1.06	1.08	1.06	1.06	1.07	1.06
	96	256	1.02	1.00	1.01	1.01	1.03	1.01	1.02	1.01
	256	32	0.84	0.86	0.83	0.86	0.92	0.94	0.92	0.94
	256	64	0.91	0.94	0.90	0.93	1.01	0.96	1.02	0.96
	256	96	0.93	0.98	0.93	0.97	1.00	0.97	1.00	0.97
	256	256	1.00	0.98	1.00	0.97	1.02	1.00	1.02	1.00
DTRMM	M	N	SIDE, TRANS				SIDE, TRANS			
			'L','N'	'L','T'	'R','N'	'R','T'	'L','N'	'L','T'	'R','N'	'R','T'
	32	256	0.64	0.67	0.88	0.92	0.69	0.70	0.83	0.84
	64	256	0.64	0.67	0.88	0.88	0.71	0.72	0.89	0.88
	96	256	0.76	0.79	0.88	0.89	0.74	0.76	0.87	0.87
	256	32	0.88	0.92	0.74	0.69	0.82	0.87	0.69	0.68
	256	64	0.87	0.89	0.71	0.72	0.88	0.90	0.77	0.75
	256	96	0.86	0.89	0.86	0.86	0.86	0.88	0.80	0.78
	256	256	0.87	0.88	0.89	0.88	0.88	0.89	0.89	0.88
DTRSM	M	N	SIDE, TRANS				SIDE, TRANS			
			'L','N'	'L','T'	'R','N'	'R','T'	'L','N'	'L','T'	'R','N'	'R','T'
	32	256	0.59	0.61	0.84	0.88	0.62	0.66	0.81	0.81
	64	256	0.61	0.63	0.84	0.85	0.65	0.69	0.87	0.86
	96	256	0.73	0.76	0.84	0.85	0.69	0.73	0.84	0.84
	256	32	0.85	0.90	0.62	0.63	0.79	0.87	0.65	0.65
	256	64	0.85	0.88	0.63	0.66	0.85	0.89	0.72	0.70
	256	96	0.85	0.87	0.78	0.80	0.83	0.86	0.76	0.75
	256	256	0.86	0.86	0.84	0.85	0.85	0.87	0.86	0.85

Table 10: GEMM-ratios for the IBM ESSL library on IBM SP2 thin node and wide node. Underlying routines: ESSL.

Routine	Dimensions		IBM SP2 thin node				IBM SP2 wide node			
DSYMM	`M`	`N`	`SIDE, UPLO`				`SIDE, UPLO`			
			'L','U'	'L','L'	'R','U'	'R','L'	'L','U'	'L','L'	'R','U'	'R','L'
	32	256	0.99	0.99	0.66	0.67	0.99	0.99	0.79	0.78
	64	256	0.96	0.96	0.82	0.83	0.99	0.99	0.89	0.89
	96	256	0.95	0.95	0.88	0.88	1.04	1.02	0.92	0.92
	256	32	0.67	0.67	0.96	0.99	0.74	0.73	0.99	0.99
	256	64	0.79	0.79	0.99	0.96	0.85	0.85	0.99	0.99
	256	96	0.85	0.86	0.96	0.94	0.91	0.90	0.99	0.99
	256	256	0.95	0.95	0.96	0.96	0.98	0.98	0.97	0.97
DSYRK	`N`	`K`	`UPLO, TRANS`				`UPLO, TRANS`			
			'U','N'	'U','T'	'L','N'	'L','T'	'U','N'	'U','T'	'L','N'	'L','T'
	32	256	1.07	1.22	1.10	1.21	0.42	0.38	0.41	0.37
	64	256	1.22	1.30	1.20	1.28	0.51	0.49	0.50	0.49
	96	256	0.85	0.87	0.82	0.83	0.58	0.57	0.58	0.57
	256	32	1.03	1.03	1.02	1.02	0.71	0.72	0.71	0.71
	256	64	1.06	1.04	1.04	1.05	0.74	0.75	0.75	0.75
	256	96	0.86	0.84	0.84	0.82	0.76	0.76	0.76	0.77
	256	256	0.89	0.87	0.87	0.86	0.76	0.76	0.76	0.76
DSYR2K	`N`	`K`	`UPLO, TRANS`				`UPLO, TRANS`			
			'U','N'	'U','T'	'L','N'	'L','T'	'U','N'	'U','T'	'L','N'	'L','T'
	32	256	0.99	1.00	1.01	0.99	0.96	0.96	0.96	0.95
	64	256	0.99	0.99	0.99	0.99	1.07	1.05	1.07	1.05
	96	256	0.99	0.98	0.98	0.98	1.02	0.99	1.02	0.99
	256	32	0.76	0.74	0.77	0.74	0.72	0.74	0.71	0.73
	256	64	0.87	0.84	0.87	0.84	0.89	0.84	0.88	0.84
	256	96	0.92	0.88	0.92	0.88	0.91	0.87	0.90	0.87
	256	256	1.01	0.96	0.92	0.96	1.00	0.96	1.00	0.96
DTRMM	`M`	`N`	`SIDE, TRANS`				`SIDE, TRANS`			
			'L','N'	'L','T'	'R','N'	'R','T'	'L','N'	'L','T'	'R','N'	'R','T'
	32	256	1.18	1.12	0.98	0.93	0.61	0.63	0.61	0.61
	64	256	1.13	1.09	1.17	1.15	0.64	0.65	0.82	0.81
	96	256	0.84	0.82	1.07	1.07	0.70	0.72	0.76	0.75
	256	32	0.92	1.01	1.15	1.15	0.56	0.59	0.68	0.68
	256	64	1.06	1.11	1.15	1.12	0.76	0.79	0.70	0.71
	256	96	0.98	1.02	0.84	0.84	0.71	0.73	0.76	0.76
	256	256	0.94	0.93	0.96	0.96	0.82	0.83	0.86	0.84
DTRSM	`M`	`N`	`SIDE, TRANS`				`SIDE, TRANS`			
			'L','N'	'L','T'	'R','N'	'R','T'	'L','N'	'L','T'	'R','N'	'R','T'
	32	256	1.13	1.09	0.97	1.00	0.59	0.61	0.61	0.60
	64	256	1.12	1.07	1.14	1.23	0.62	0.64	0.82	0.80
	96	256	0.84	0.82	1.07	1.16	0.67	0.72	0.76	0.75
	256	32	1.04	1.11	1.14	1.14	0.57	0.61	0.68	0.69
	256	64	1.19	1.21	1.08	1.10	0.76	0.79	0.70	0.71
	256	96	1.11	1.11	0.82	0.84	0.71	0.73	0.75	0.75
	256	256	0.93	0.92	0.94	0.95	0.80	0.82	0.85	0.84

Table 11: GEMM-ratios on a single node of the Intel PARAGON. Underlying routines: Paragon Basic Math Library and KD-GEMV.

Routine	Dimensions		Intel PARAGON			
DSYMM	`M`	`N`	`SIDE, UPLO`			
			'L','U'	'L','L'	'R','U'	'R','L'
	32	256	1.84	1.86	1.12	1.13
	64	256	1.54	1.56	1.09	1.09
	96	256	1.39	1.40	1.08	1.08
	256	32	1.05	1.05	1.24	1.26
	256	64	1.10	1.10	1.16	1.16
	256	96	1.12	1.12	1.11	1.11
	256	256	1.12	1.12	1.06	1.06
DSYRK	`N`	`K`	`UPLO, TRANS`			
			'U','N'	'U','T'	'L','N'	'L','T'
	32	256	1.16	0.61	1.17	0.62
	64	256	0.99	0.70	1.04	0.71
	96	256	0.99	0.73	1.07	0.75
	256	32	1.04	1.08	1.09	1.12
	256	64	0.98	0.91	1.56	0.97
	256	96	0.99	0.89	1.38	0.93
	256	256	0.99	0.87	1.15	0.88
DSYR2K	`N`	`K`	`UPLO, TRANS`			
			'U','N'	'U','T'	'L','N'	'L','T'
	32	256	2.86	1.33	2.90	1.33
	64	256	2.04	1.15	2.13	1.16
	96	256	1.72	1.08	1.84	1.10
	256	32	1.11	0.98	2.30	1.08
	256	64	1.21	1.00	1.87	1.06
	256	96	1.24	1.01	1.69	1.06
	256	256	1.26	1.03	1.44	1.04
DTRMM	`M`	`N`	`SIDE, TRANS`			
			'L','N'	'L','T'	'R','N'	'R','T'
	32	256	0.76	0.98	1.08	1.28
	64	256	0.69	0.89	0.98	1.07
	96	256	0.72	0.86	0.93	0.99
	256	32	0.80	0.88	0.60	0.61
	256	64	0.81	0.91	0.69	0.68
	256	96	0.80	0.91	0.75	0.74
	256	256	0.80	0.92	0.89	0.90
DTRSM	`M`	`N`	`SIDE, TRANS`			
			'L','N'	'L','T'	'R','N'	'R','T'
	32	256	0.77	0.76	1.12	1.28
	64	256	0.71	0.78	0.99	1.07
	96	256	0.72	0.78	0.93	0.98
	256	32	0.79	0.85	0.61	0.61
	256	64	0.80	0.87	0.69	0.67
	256	96	0.79	0.87	0.74	0.73
	256	256	0.79	0.88	0.88	0.89

Table 12: GEMM-ratios for machine specific libraries on SGI Indy with MIPS R4000 and R4400 processor. Underlying routines: SGI library `libblas`.

Routine	Dimensions		SGI Indy (R4000)				SGI Indy (R4400)			
DSYMM	M	N	SIDE, UPLO				SIDE, UPLO			
			'L','U'	'L','L'	'R','U'	'R','L'	'L','U'	'L','L'	'R','U'	'R','L'
	32	256	1.21	1.24	1.10	1.06	1.27	1.29	1.06	1.07
	64	256	1.25	1.28	1.14	1.16	1.27	1.31	1.09	1.11
	96	256	1.26	1.30	1.16	1.18	1.26	1.31	1.12	1.13
	256	32	1.01	1.03	1.44	1.45	1.02	0.99	1.39	1.35
	256	64	1.14	1.13	1.34	1.32	1.16	1.15	1.29	1.29
	256	96	1.16	1.18	1.30	1.28	1.18	1.18	1.27	1.26
	256	256	1.20	1.22	1.21	1.23	1.22	1.24	1.19	1.20
DSYRK	N	K	UPLO, TRANS				UPLO, TRANS			
			'U','N'	'U','T'	'L','N'	'L','T'	'U','N'	'U','T'	'L','N'	'L','T'
	32	256	1.42	1.43	1.34	1.43	1.46	1.55	1.45	1.49
	64	256	1.24	1.23	1.20	1.23	1.24	1.26	1.25	1.27
	96	256	1.16	1.16	1.14	1.17	1.21	1.23	1.20	1.23
	256	32	1.12	1.01	1.11	0.96	1.13	1.07	1.12	1.06
	256	64	1.06	1.05	1.05	1.08	1.12	1.11	1.11	1.11
	256	96	1.05	1.09	1.03	1.08	1.12	1.11	1.11	1.11
	256	256	1.03	1.08	1.01	1.08	1.10	1.12	1.10	1.11
DSYR2K	N	K	UPLO, TRANS				UPLO, TRANS			
			'U','N'	'U','T'	'L','N'	'L','T'	'U','N'	'U','T'	'L','N'	'L','T'
	32	256	1.83	1.89	1.82	1.86	1.69	1.88	1.69	1.87
	64	256	1.54	1.57	1.50	1.56	1.43	1.58	1.44	1.59
	96	256	1.41	1.41	1.41	1.42	1.36	1.44	1.36	1.44
	256	32	1.11	1.06	1.11	1.04	1.09	1.05	1.09	1.04
	256	64	1.15	1.13	1.15	1.12	1.15	1.15	1.15	1.15
	256	96	1.17	1.15	1.18	1.16	1.17	1.18	1.17	1.18
	256	256	1.17	1.20	1.23	1.21	1.21	1.22	1.21	1.22
DTRMM	M	N	SIDE, TRANS				SIDE, TRANS			
			'L','N'	'L','T'	'R','N'	'R','T'	'L','N'	'L','T'	'R','N'	'R','T'
	32	256	0.68	0.69	0.96	0.95	0.78	0.79	0.95	0.94
	64	256	0.81	0.76	0.99	0.96	0.88	0.85	0.94	0.94
	96	256	0.81	0.80	0.99	0.98	0.92	0.86	0.96	0.97
	256	32	0.89	0.86	1.02	0.98	0.95	0.91	0.95	0.81
	256	64	0.85	0.85	1.01	1.00	0.93	0.90	0.95	0.88
	256	96	0.86	0.87	1.01	1.00	0.94	0.91	0.98	0.93
	256	256	0.85	0.85	1.00	1.00	0.94	0.90	0.99	0.98
DTRSM	M	N	SIDE, TRANS				SIDE, TRANS			
			'L','N'	'L','T'	'R','N'	'R','T'	'L','N'	'L','T'	'R','N'	'R','T'
	32	256	0.77	0.85	1.02	0.96	0.90	0.91	0.91	0.88
	64	256	0.86	0.88	1.01	0.95	0.96	0.95	0.92	0.89
	96	256	0.88	0.89	1.00	0.95	0.98	0.93	0.96	0.91
	256	32	0.91	0.91	1.01	0.99	0.97	0.91	1.03	0.97
	256	64	0.87	0.91	1.01	1.00	0.94	0.91	1.00	0.97
	256	96	0.88	0.92	0.99	0.98	0.96	0.92	1.01	0.94
	256	256	0.86	0.88	0.99	0.94	0.96	0.90	0.99	0.92

Table 13: GEMM-ratios on a single node of the Parsytec GC/PP for the original level 3 BLAS model implementations from netlib. Underlying routines: BP-DGEMM and original netlib BLAS (second column), BP-DGEMM, POL-DGEMV and original netlib BLAS (third column).

Routine	Dimensions		Parsytec (DGEMV)				Parsytec (POL-DGEMV)			
DSYMM	`M`	`N`	`SIDE, UPLO`				`SIDE, UPLO`			
			`'L','U'`	`'L','L'`	`'R','U'`	`'R','L'`	`'L','U'`	`'L','L'`	`'R','U'`	`'R','L'`
	32	256	1.98	1.99	4.62	4.64	2.09	2.10	4.57	4.57
	64	256	2.05	2.06	4.23	4.22	2.06	2.06	4.21	4.18
	96	256	2.84	2.82	4.17	4.15	2.85	2.83	4.14	4.12
	256	32	2.26	2.25	3.20	3.20	2.27	2.26	3.02	3.01
	256	64	2.56	2.54	3.52	3.51	2.58	2.57	3.50	3.50
	256	96	2.68	2.67	3.85	3.85	2.70	2.69	3.82	3.82
	256	256	2.80	2.80	4.02	4.01	2.82	2.82	3.98	3.97
DSYRK	`N`	`K`	`UPLO, TRANS`				`UPLO, TRANS`			
			`'U','N'`	`'U','T'`	`'L','N'`	`'L','T'`	`'U','N'`	`'U','T'`	`'L','N'`	`'L','T'`
	32	256	1.24	0.76	1.27	0.77	1.93	1.15	1.98	1.12
	64	256	1.57	1.08	1.62	1.11	2.20	1.52	2.31	1.45
	96	256	1.81	1.33	1.85	1.36	2.40	1.79	2.49	1.79
	256	32	2.24	2.04	2.29	2.08	2.58	2.38	2.71	2.38
	256	64	2.41	1.98	2.48	2.04	2.75	2.30	2.88	2.27
	256	96	2.55	2.05	2.61	2.11	2.97	2.42	3.09	2.41
	256	256	2.58	1.98	2.63	2.05	2.99	2.32	3.11	2.34
DSYR2K	`N`	`K`	`UPLO, TRANS`				`UPLO, TRANS`			
			`'U','N'`	`'U','T'`	`'L','N'`	`'L','T'`	`'U','N'`	`'U','T'`	`'L','N'`	`'L','T'`
	32	256	4.17	2.33	4.12	2.36	4.14	2.41	4.07	2.36
	64	256	3.34	2.37	3.31	2.41	3.31	2.45	3.28	2.41
	96	256	3.19	2.46	3.16	2.51	3.17	2.54	3.15	2.51
	256	32	2.26	2.72	2.26	2.76	2.26	2.79	2.27	2.76
	256	64	2.48	2.59	2.47	2.63	2.46	2.67	2.48	2.63
	256	96	2.64	2.68	2.64	2.71	2.62	2.76	2.63	2.72
	256	256	2.84	2.57	2.84	2.63	2.82	2.66	2.83	2.64
DTRMM	`M`	`N`	`SIDE, TRANS`				`SIDE, TRANS`			
			`'L','N'`	`'L','T'`	`'R','N'`	`'R','T'`	`'L','N'`	`'L','T'`	`'R','N'`	`'R','T'`
	32	256	0.87	0.65	2.80	2.50	1.10	0.86	3.16	2.74
	64	256	1.15	0.89	2.69	2.59	1.43	1.13	3.01	2.90
	96	256	1.87	1.53	2.73	2.70	2.37	1.98	3.08	3.03
	256	32	2.54	2.15	1.10	1.12	2.84	2.45	1.37	1.42
	256	64	2.72	2.26	1.48	1.51	3.06	2.59	1.90	1.93
	256	96	2.85	2.36	1.79	1.80	3.23	2.73	2.28	2.29
	256	256	2.92	2.40	2.66	2.69	3.32	2.76	3.00	3.03
DTRSM	`M`	`N`	`SIDE, TRANS`				`SIDE, TRANS`			
			`'L','N'`	`'L','T'`	`'R','N'`	`'R','T'`	`'L','N'`	`'L','T'`	`'R','N'`	`'R','T'`
	32	256	1.10	0.74	2.73	2.46	1.41	0.91	3.07	2.67
	64	256	1.37	0.95	2.60	2.53	1.69	1.19	2.89	2.79
	96	256	2.09	1.55	2.64	2.63	2.63	1.94	2.95	2.91
	256	32	2.78	2.14	0.99	0.98	3.11	2.44	1.21	1.25
	256	64	2.97	2.25	1.38	1.40	3.34	2.56	1.70	1.72
	256	96	3.11	2.34	1.68	1.69	3.52	2.68	2.08	2.09
	256	256	3.18	2.38	2.56	2.61	3.61	2.72	2.87	2.90

Table 14: Multiprocessor performance of the GEMM-Based DSYR2K on the ALLIANT FX/2816.

Dimensions			DSYR2K			DGEMM	
K	N	p	Mflops	S_p	E_p	Mflops	E_{GEMM}
512	512	1	34.76	1.0	1.0	36.81	0.94
		2	62.50	1.8	0.9	70.48	0.82
		4	106.11	3.0	0.8	129.93	0.82
		6	129.81	3.7	0.6	179.13	0.73
32	512	1	25.72	1.0	1.0	28.11	0.92
		2	38.89	1.5	0.8	51.82	0.75
		4	53.03	2.1	0.5	87.93	0.60
		6	55.60	2.2	0.4	108.98	0.51
512	32	1	23.29	1.0	1.0	28.11	0.92
		2	33.82	1.5	0.7	33.70	1.00
		4	43.30	1.9	0.5	40.76	1.06
		6	42.67	1.8	0.3	42.41	1.00
128	128	1	27.71	1.0	1.0	32.99	0.84
		2	43.51	1.6	0.8	58.94	0.73
		4	61.38	2.2	0.6	97.44	0.63
		6	64.12	2.3	0.4	126.88	0.50

Table 15: Performance in Mflops for NEC SX-3. Leading dimension of arrays is 512.

Dimensions		SSYMM	SSYRK	SSYR2K	STRMM	STRSM
M(N)	N(K)	'L','U'	'U','N'	'U','N'	'L','N'	'L','N'
16	512	266.3	85.3	105.5	142.4	125.3
32	512	643.9	192.7	207.3	406.3	368.4
64	512	1416.8	409.5	410.6	864.1	808.4
128	256	2229.8	824.1	813.1	1101.0	1236.7
512	16	716.9	1090.1	1254.6	652.3	536.1
512	32	1109.1	767.2	729.7	935.4	913.1
512	64	1527.6	817.6	779.9	1194.2	1404.6
512	128	1883.0	825.3	803.2	1384.7	1880.0
512	512	2284.5	829.3	825.0	1428.5	2008.9

Table 16: Performance in Mflops for NEC SX-3. Leading dimension of arrays are 530.

Dimensions		SSYMM	SSYRK	SSYR2K	STRMM	STRSM
M(N)	N(K)	'L','U'	'U','N'	'U','N'	'L','N'	'L','N'
16	512	271.5	85.0	335.5	142.0	124.8
32	512	646.6	192.0	711.8	405.4	366.2
64	512	1417.0	409.3	1463.2	862.2	807.6
128	256	2234.2	824.0	2251.7	1102.9	1240.5
512	16	1048.5	1107.7	1279.8	655.2	533.8
512	32	1468.0	1383.8	1696.8	948.1	908.0
512	64	1838.1	1579.4	2031.2	1210.2	1393.6
512	128	2106.0	1607.7	2210.8	1392.9	1865.7
512	512	2363.3	1628.4	2373.0	1429.5	1986.9

Table 17: Benchmark results for canonical input file DMARK01

Machines	GEMM-based	Original netlib	Vendor supplied	DGEMM Mflops	Comments
IBM RS6K 530H	0.73	0.34	0.92	51.2	ESSL underlying routines
IBM RS6K 530H	0.78	0.34	0.92	51.2	ESSL under. except POL DGEMV
IBM RS6K 250	0.77	0.30	0.91	42.2	ESSL underlying routines
IBM RS6K 250	0.80	0.30	0.91	42.2	ESSL under. except POL DGEMV
IBM SP2	0.67	0.27	0.87	163.8	single thin node
IBM SP2	0.70	0.33	0.90	197.6	single wide node
Intel PARAGON	0.72	0.20	0.71	38.5	Underlying from -lkmath
Intel PARAGON	0.75	0.20	0.71	38.5	U. f. -lkmath except KD DGEMV
Parsytec GC/PP	0.75	0.32	—	46.5	1 proc. POL DGEMV, B.P. DGEMM
SGI Indy	0.82	0.39	0.80	26.0	R4000 processor
SGI Indy	0.86	0.38	0.81	40.5	R4400 processor

Table 18: Benchmark results for canonical input file DMARK02

Machines	GEMM-based	Original netlib	Vendor supplied	DGEMM Mflops	Comments
IBM RS6K 530H	0.83	0.31	0.95	58.8	ESSL underlying routines
IBM RS6K 530H	0.87	0.31	0.95	58.8	ESSL under. except POL DGEMV
IBM RS6K 250	0.81	0.28	0.95	48.8	ESSL underlying routines
IBM RS6K 250	0.85	0.28	0.95	48.8	ESSL under. except POL DGEMV
IBM SP2	0.74	0.22	0.92	197.1	single thin node
IBM SP2	0.83	0.30	0.95	227.6	single wide node
Intel PARAGON	0.79	0.18	0.84	44.6	Underlying from -lkmath
Intel PARAGON	0.82	0.18	0.84	44.6	U. f. -lkmath except KD DGEMV
Parsytec GC/PP	0.81	0.29	—	53.1	1 proc. POL DGEMV, B.P. DGEMM
SGI Indy	0.81	0.34	0.78	30.9	R4000 processor
SGI Indy	0.85	0.33	0.79	48.1	R4400 processor