# Local Basic Linear Algebra Subroutines (LBLAS) for Distributed Memory Architectures and Languages with Array Syntax

# Local Basic Linear Algebra Subroutines (LBLAS) for Distributed Memory Architectures and Languages with Array Syntax

S. Lennart Johnsson
Luis F. Ortiz

TR-09-92

April 1992
Revised September 1992


Parallel Computing Research Group

Center for Research in Computing Technology
Harvard University
Cambridge, Massachusetts

To appear in the *International Journal of Supercomputer Applications.*

# Local Basic Linear Algebra Subroutines (LBLAS) for distributed memory architectures and languages with array syntax

S. Lennart Johnsson[1] and Luis F. Ortiz
Thinking Machines Corp.
245 First Street
Cambridge, MA 02142
Johnsson@think.com, Ortiz@think.com

## Abstract

We describe a subset of the level–1, level–2, and level–3 BLAS implemented for each node of the Connection Machine system CM–200. The routines, collectively called LBLAS, have interfaces consistent with languages with an array syntax such as Fortran 90. One novel feature, important for distributed memory architectures, is the capability of performing computations on multiple instances of objects in a single call. The number of instances and their allocation across memory units, and the strides for the different axes within the local memories, are derived from an array descriptor that contains type, shape, and data distribution information. Another novel feature of the LBLAS is a selection of loop order for rank–1 updates and matrix–matrix multiplication based upon array shapes, strides, and DRAM page faults. The peak efficiencies for the routines are in excess of 75%. Matrix–vector multiplication achieves a peak efficiency of 92%. The optimization of loop ordering has a success rate exceeding 99.8% for matrices for which the sum of the lengths of the axes is at most 60. The success rate is even higher for all possible matrix shapes. The performance loss when a nonoptimal choice is made is less than $\sim$15% of peak and typically less than 1% of peak. We also show that the performance gain for high rank updates may be as much as a factor of 6 over rank–1 updates.

# 1 Introduction

The Basic Linear Algebra Subroutines [1, 2, 8] (BLAS) are used in many scientific codes, often being critical for the performance of those codes. For many computer architectures, implementations in low–level languages have been made to ensure a desirable efficiency, though improved compiler technologies recently have allowed BLAS for some architectures to be coded in high–level languages, suitably structured, without significant loss of efficiency [3, 7, 10]. We discuss the issues involved in designing local BLAS for distributed memory architectures, programmed in languages with an array syntax. We report on the

[1] Also affiliated with the Division of Applied Sciences, Harvard University

techniques used and the performance achieved in a subset of the BLAS for each node of a Connection Machine system CM–200, a distributed memory architecture with up to 2048 processing nodes. The techniques are applicable to many distributed memory architectures and are now being applied in developing optimized BLAS for the Connection Machine system CM–5 [15]. The subset of the BLAS described here, referred to as LBLAS for Local BLAS, forms part of the Connection Machine Scientific Software Library, CMSSL [18]. The LBLAS can be accessed directly, or indirectly, through the CMSSL Distributed BLAS (DBLAS) [11]. Indeed, there is only one interface. The distribution of the arrays determines whether calls to the LBLAS suffice to accomplish the desired operation, or if a distributed algorithm calling both communication routines and the LBLAS shall be invoked. The LBLAS are used for both dense and block sparse matrix operations, and are used as well in the CMSSL linear system solvers [9] and eigenanalysis routines. Frequently, the LBLAS are also used directly in solving partial differential equations.

The BLAS have emerged over time, starting with level–1 BLAS [8] for vector operations such as inner products (_DOT), scaling of a vector (_SCAL), and the addition of a scaled vector to another vector (_AXPY). The underscore represents a symbol denoting the relevant data type (S,D,C,Z). Operations such as matrix–vector multiplication can be expressed in terms of level–1 BLAS. But, the number of memory references may be excessive. Scaling a vector requires two memory accesses for a single floating–point operation, assuming that the constant is available in a register. An inner product requires two memory accesses for each pair of floating–point operations (plus a store of the result), while an _AXPY operation requires three memory operations per pair of floating–point operations, assuming the constant is available in a register.

The level–2 BLAS [2] were defined to allow for increased utilization of arithmetic units in architectures with a higher data motion capacity between registers, or a cache, and the arithmetic units, than between the registers or the cache and main memory. For instance, in matrix–vector multiplication $y \leftarrow Ax + y$, the vector $x$ can be loaded into a register file, followed by the sequence of operations $z(:) \leftarrow x(i)A(:,i) + z(:)$ for $i = 1, 2, \ldots, Q$ for a matrix of shape $P \times Q$. Each iteration is an _AXPY operation on vectors of length $P$. The _AXPY operation requires three memory references per element when $z$ is read from and stored in memory for each iteration. However, if $z$ is kept in the register file until completion, then each iteration only requires one read from memory for each pair of arithmetic operations. The level–2 BLAS include matrix–vector multiplication as a primitive, allowing extraneous memory references to be avoided without relying on interprocedural analysis by the compiler. Commercial compilers still rarely achieve close to peak performance even on single routines written in a completely architecturally independent way. However, by a suitable partitioning, unrolling, and possibly skewing of loops by the programmer, state–of–the–art compilers for some architectures produce very efficient code [3, 7, 10]. But, on many architectures, assembly level programming is still required to achieve the desired level of efficiency in using registers, caches, memory and pipelines. The CM–200 belong to this category of computer systems. In architectures with a single data path to memory, such as the CM–200, the level–2 BLAS offer a potential speedup for matrix–vector multiplication by a factor of two compared to a _DOT based

| Operation | Operations per mem. ref. |
|---|---|
| _SCAL | $\frac{1}{2}$ |
| _AXPY | $\frac{2}{3}$ |
| _DOT | 1 |
| _GER | 1 |
| _GEMV | 2 |
| _GEMM | b |

Table 1: Floating–point operations per memory reference for a few BLAS functions.

algorithm, and a factor of three compared to an _AXPY based algorithm.

The level–3 BLAS [1] allow computations, such as matrix–matrix multiplication, to be performed with less demand on the memory bandwidth than when level–2 BLAS routines are used in the absence of interprocedural analysis and subsequent optimization of memory references by a compiler. The matrix multiplication $C \leftarrow A \times B + D$ may be performed as a sequence of multiplications of $b$ by $b$ subblocks. If the blocks required for a block matrix multiplication fit in the registers, then $2b^3$ floating–point operations can be carried out using $3b^2$ memory references for data input and $b^2$ memory references for data output. Furthermore, if all contributions to a block of $C$ are accumulated in the registers, i.e., only the final result is stored in memory, then only $2b^2$ elements need to be loaded from memory for each set of $2b^3$ floating–point operations. Delaying all stores until the computations for a $b \times b$ block of $C$ are completed results in $b$ floating–point operations per memory reference, on average.

Note that for matrix–matrix multiplication, an algorithm based on the level–1 BLAS is memory bandwidth limited when there is a single floating–point unit for each data path to memory and the data paths internal to the floating–point processor and the paths to memory are of the same width. An algorithm based on level–2 BLAS is balanced with respect to the demand for memory bandwidth and computational capability, and an algorithm based on the level–3 BLAS is limited by the arithmetic processing capability. Table 1 summarizes the operations count per memory reference for some functions in the level–1 BLAS.

Throughout this paper, in the operation $C \leftarrow A \times B$, the matrix $A$ is of shape $P \times Q$, $B$ of shape $Q \times R$, and $C$ of shape $P \times R$. For matrix–vector multiplication $R = 1$, the vector $x$ replaces $B$, and the vector $y$ replaces $C$. For outer products $Q = 1$, the $x$ replaces $A$ and $y^T$ replaces $B$.

We only discuss BLAS for real data types. No particular optimization is currently performed in the CMSSL LBLAS for complex data types. For complex data types real routines are called as required.

The functionality of the LBLAS we have implemented deviates somewhat from the corresponding subset of the conventional BLAS [1, 2, 8] in that all scaling factors in LBLAS

are assumed to be one. For instance, for matrix–vector multiplication, _GEMV supports the operation $y \leftarrow \alpha Ax + \beta y$, where $\alpha$ and $\beta$ are scalars, whereas the CMSSL local matrix–vector multiplication performs the operation $y \leftarrow Ax + y$. The reason that, at the moment, the LBLAS in CMSSL assume that $\alpha = \beta = 1$ is for compatibility with the CMSSL DBLAS. The functionality of any routine in the CMSSL is independent of the data distribution, and BLAS are no exception. For the DBLAS, mixing array arguments of different rank may have severe performance implications, unless the required data motion for aligning the arrays is efficiently implemented. When the data motion issues for mixing arrays of different rank are satisfactorily solved for the DBLAS, then the CMSSL LBLAS will be extended to the exact same functionality as the corresponding routine in the conventional BLAS. For a discussion of the data motion issues in the DBLAS see [4, 5, 6, 11].

In the following, for the convenience of the reader, we discuss the LBLAS using the traditional BLAS names whenever the distinction is either irrelevant or clear from the context. Whenever there is a need to stress that the discussion refers to the LBLAS, we prefix the BLAS names with CMSSL, such as, for instance, CMSSL_DDOT for the CMSSL routine computing inner products in double precision. For the actual names of the CMSSL routines, see [18]. The names of the CMSSL routines, and hence the LBLAS, are not limited to six characters, and do not have the data type encoded in the routine name. All CMSSL routines have interfaces consistent with a language with an array syntax such as Fortran 90 [12], Connection Machine Fortran [17], and the emerging High Performance Fortran standard. BLAS interfaces are consistent with Fortran 77 syntax. The parts of the LBLAS that execute in each floating–point unit of the CM–200 are mostly implemented in assembly code, with some parts, however, in microcode. The parts of the code that execute on the Front–End computer is in higher–level code, such as C, Fortran, or Lisp.

In Section 2, we discuss some of the design issues for high performance library routines for languages with an array syntax and distributed memory multiprocessors. In Section 3 we discuss the relevant architectural features of the CM–200. Section 4 discusses our level–1 LBLAS, while level–2 LBLAS are presented in Section 5. In Section 6 we discuss how the loop ordering for matrix–matrix multiplication and outer product computation is determined at run–time. A summary follows in Section 7.

# 2 Library issues for languages with an array syntax

The LBLAS are designed to support concurrent operations on distributed data structures and languages with an array syntax. These facts have a fundamental impact on the interfaces of the routines with respect to: 1) what information is provided explicitly through arguments to maintain consistency with the languages, 2) what information is provided through arguments in order to realize a desired level of efficiency in execution, and 3) the amount of concurrency handled by the routines.

One of the unique features of the CMSSL is the ability of routines to perform compu-

tations on multiple instances of objects in the same call, with instances embedded in a multidimensional array. For example, in the solution of the Navier–Stokes equation by a finite difference technique, matrix–vector multiplication is required in each grid point [13]. The matrix in a grid point defines one instance of the grid point matrices. For a regular three–dimensional discretization of the domain, three additional array axes may be used to identify the matrices in different grid points. Similarly, in solving partial differential equations by the finite element technique, there is a matrix associated with each element. For a regular discretization of a three-dimensional domain, three additional array axes may be used to identify each elemental matrix, just as in the finite difference case. For unstructured discretizations, a single axis in addition to the axes defining each elemental matrix can be used to enumerate the elemental matrices. In a traditional BLAS, the looping over instances is made by placing the call inside a loop or a set of nested loops. In the CMSSL, the entire array is used as an argument in the subroutine call, and the looping over instances handled is inside the subroutine. The multiple–instance capability of the routines allows extraneous data motion as well as unnecessary temporary storage allocation to be avoided. This capability also provides additional opportunities to optimize scheduling of operations compared to single–instance routines (without powerful interprocedural analysis).

With respect to performance, it is highly desirable to minimize any unnecessary data motion, in particular that between processing nodes. Depending upon the data distribution rules used by the compiler and the run–time system, and the techniques used for dealing with computations on multiple instances of (sub)arrays, limiting each call to the BLAS to a single instance may incur a substantial performance penalty [4]. Thus, in order to assure a minimum amount of data motion, avoiding the creation of temporary arrays representing sections of arrays, (in particular the creation of temporary arrays with a potentially significantly different data distribution compared to the original array from which they may be extracted), and maximizing the optimization opportunities in scheduling operations, the LBLAS, like all CMSSL routines, are based on passing entire arrays in place. No data motion is associated with the subroutine call itself. The decision of which instances are treated concurrently, and which ones are treated sequentially, is made within each routine of the CMSSL.

In the LBLAS for the CM-200, instances assigned to different processing nodes are treated concurrently, while instances assigned to the same processing node are treated sequentially, one instance at a time. However, in the LBLAS currently under development for the CM–5, whether the looping over multiple instances in a processing node is made in an outer or inner loop depends upon which loop order is predicted to yield the best performance. A mechanism for optimal loop ordering at run–time, including the instance axes, is particularly important when DRAM page faults may affect the performance significantly. How the multiple–instance capability designed into the LBLAS is exploited for the CM–5 will be described elsewhere. The scheduling order is transparent with respect to the call. The interface is unaffected. Thus, programs calling the LBLAS or the DBLAS are portable between the CM–200 and the CM–5. Though instances in a processing node are treated sequentially on the CM–200, the multiple–instance capability still implies significant performance advantages due to reduced data motion, in particular,

for computations on many small objects, due to reduced overhead by amortizing over all instances the cost of determining and setting up shared information, such as strides within an instance and the calling sequence to be used for each instance.

From an implementation point of view, one important consequence of the LBLAS multiple–instance capability, is that no implicit assumption can be made about either the absolute values or the relative values of the strides for the different axes defining an instance, the *problem axes (axis)*. The problem axes in the CMSSL can be chosen arbitrarily from a multidimensional array passed in–place. Hence, for a given data distribution across the memory units, the stride within a row may be larger ( or smaller) than the stride within a column depending upon the relative ordering of the row and column axes in the array in which the instances are embedded. Moreover, on the Connection Machine systems, the lengths of the axes' segments assigned to a memory unit, and hence the strides, are not known until run–time. Data distribution is a run–time system function, allowing the same user program to be executed on a different number of processing nodes without recompilation.

The form of a call to the LBLAS is illustrated by the following example:

**Array** y(N,M,K), x(N,K,L), A(M,L,N,K)

**gen_matrix_vect_mult**(y, A, x, 2, 1, 2, 3, ier).

In the example, $y$ and $x$ represent either single vectors or (multidimensional) arrays of vectors; $A$ represents a matrix, or a (multidimensional) array of matrices. The rank of the array $A$ must be one higher than the ranks of the arrays $y$ and $x$, which are of the same rank. The number 2 succeeding $x$ states that the problem axis for $y$ is the second axes of the array $y$, i.e., the axis of extent $M$. Similarly, the number 1 states that the problem row axis for $A$ is axis 1 of the array $A$, and the problem column axis is axis 2. The shape of each instance of $A$ is $M \times L$. The problem axis for $x$ is axis 3 of the array $x$. Thus, the above call defines multiple matrix–vector multiplications. Each instance consists of the multiplication of an $M \times L$ matrix by a vector of length $L$. There are $N \times K$ such instances. The call is independent of the distribution of the arrays. However, in order for the operation to be well–defined, we require in the CMSSL that the arrays $y$, $A$, and $x$ have *conforming shapes*, i.e., that the shapes of all arrays with the problem axes excluded are identical. In the example, it is easily seen that exclusion of the problem axes results in arrays of shape $N \times K$.

Operations requiring the transpose of $A$ can be performed using the same interface. The specifications of the row and column axes of $A$ are simply interchanged.

The example interface above is consistent with a language with array syntax, such as Fortran 90 [12] or Connection Machine Fortran [17], in that the call contains no explicit information about array data types or shapes. This information is clearly needed for most BLAS. In the case of distributed data structures, it is also necessary to know how arrays are distributed over the different memory units, as well as within the memory units. On the Connection Machine systems, this information is kept in a *descriptor*. Passing an array to a subroutine, in fact, implies passing a pointer to the descriptor. In the traditional

BLAS, the data type is encoded in the name, each call only handles a single instance, and the length of the leading dimension of two–dimensional arrays is passed explicitly as an argument.

The local BLAS we have implemented support all operations of the form

$$C^{op_C} \leftarrow C^{op_c} \pm A^{op_A} \times B^{op_B},$$

where $op_A$ and $op_C$ are of the type $N$, or $T$ for normal or transpose, respectively. $op_B$ is of type $N$, $T$, $C$ or $H$ where $C$ stands for complex conjugate and $H$ for Hermitian (complex conjugate transpose). (The selection of the desired combination of operand options for the LBLAS is made through a mode parameter not shown in the subroutine call above.)

# 3 The Connection Machine system CM–200

The CM–200 [14] has up to 2048 floating–point processors interconnected via a Boolean cube network. The elements of the architecture are shown in Figure 1. A *node* consists of a floating–point processor, its local memory, and associated communication circuitry. Instructions and scalar data are allocated to the memory of the Front–End computer. Arrays are allocated across the memories of the floating–point processors. Instructions that operate on data allocated across a subset of nodes are broadcast to all nodes. Likewise, scalar values that are required in the nodes are broadcast from the Front–End computer. The result of a global reduction operation is a scalar that resides in the memory of the Front–End computer. So called segmented reduction operations, i.e., concurrent reductions on disjoint segments of the index space, result in arrays that are allocated across some subset of nodes. For a detailed description of the memory management system on the CM–200, see [16, 17]. For a description of BLAS operating on distributed data structures, see [5, 6, 11]. Here we focus on the BLAS in each node and do not further address the issues for distributed data structures and communication.

Each floating–point processor has 64–bit wide data paths, but the path to local memory is only 32–bits wide. Figure 2 illustrates the local node architecture. Each processor has one floating–point multiplier and one adder for 32–bit or 64–bit data types, 32 registers and 4 Mbytes of local memory. The clock frequency is 10 MHz. The peak floating–point rate for each processor is 20 Mflop/s, both in 32–bit and 64–bit precision. But, for operations on 64–bit data types, the peak rate is limited to 10 Mflop/s for any operation that requires one operand to be loaded from memory. The arithmetic units are pipelined, with a pipeline length of 2 cycles for the adder. The floating–point processors do not have an inner–product instruction. Our vectorized inner–product results in two partial sums that are added in scalar mode.

There is a one cycle delay for memory operations. In the Connection Machine Instruction Set (CMIS), a vectorized instruction set, most instructions require a minimum of 6 cycles. For 32–bit data types, one item can be loaded from memory every cycle. The memory

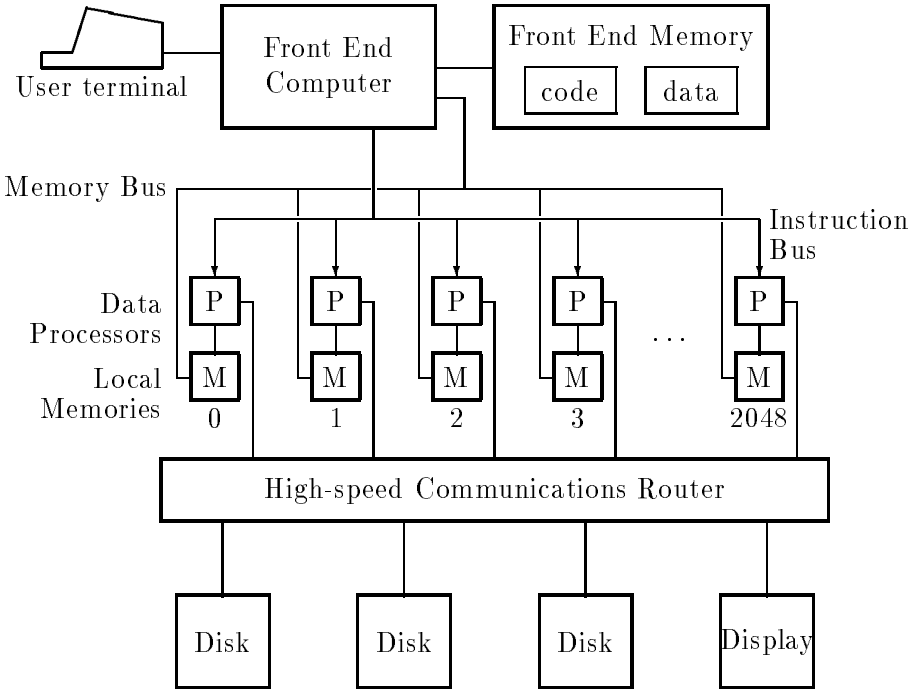

Figure 1: The Connection Machine System CM–200.

is made up of DRAM (Dynamic Random Access Memory) operated in page mode. The page size is 1 kbytes with 4 Mbit DRAM memory chips. Successive loads of data elements within the same page require a single cycle per element, while loading elements stored in different pages, incurs a page fault. Page faults add one cycle to a load of a data element. Storing data in memory requires slightly less than two cycles per item. Loads and stores of 64–bit data items require 2 and 4 cycles, respectively. DRAM page faults do not extend the time for store operations.

For 32–bit data types, matrix–vector multiplication achieves a good balance between the available memory bandwidth and the processing capability of each floating–point unit. However, for 64–bit operands, the difference in the width of the data path internal to a processor and the data path to memory, suggests a level–3 BLAS for matrix–matrix multiplication. But, because of the limited number of registers and the pipeline delays, matrix–vector multiplication is used for matrix–matrix multiplication also on 64–bit data types. Which lower level BLAS are used and the loop order, may have a significant impact on performance. Our matrix–matrix multiplication routines estimate the performance expected with different loop orders, accounting for the impact of different vector lengths, various overheads, the number of DRAM page faults, and vector chaining.

# 4 Level–1 LBLAS

The estimated peak performance rates for the level–1 LBLAS we have implemented are given in Table 2, together with the measured performance for arrays with sizes covering a large fraction of the range possible in each CM–200 node. The estimated peak rates are

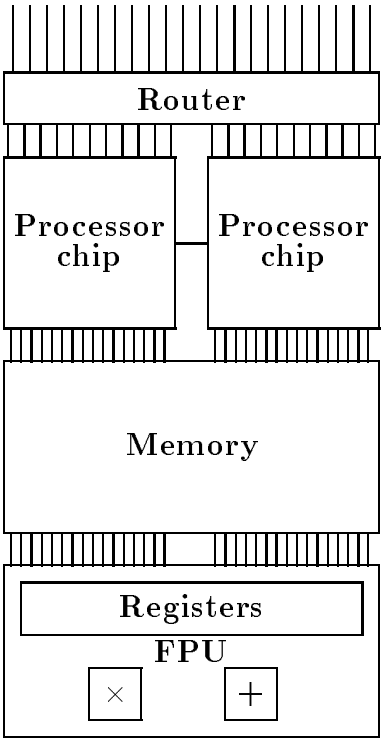


Figure 2: A CM–200 processing node.

computed based on the required number of memory references for the operation. A single load from memory is required for each pair of floating–point operations in computing $L_2$ norms (_NRM2), while two loads are required for a pair of floating–point operations in computing inner products (_DOT). Two loads and one store are required for _AXPY operations. Load/store operations in 64–bit precision require twice as many cycles as in 32–bit precision.

The estimated peak rate quantifies the feasibility of the architecture for the stated operations. We measure the quality of the implementation, i.e., the algorithm, the program structure, and the instruction set, by the achieved *efficiency*, defined as the realized performance as a percentage of the estimated peak performance. The efficiencies for various array sizes can be determined from Table 2. The peak efficiencies are: 80% for CMSSL_DNRM2, 85% for CMSSL_DDOT, and 100% for CMSSL_DAXPY. The last figure indicates that stores do not require a full two cycles. The reason that the efficiency of the CMSSL_DAXPY routine is higher than that of the CMSSL_DDOT routine is largely due to the inefficiency in the final accumulation in our vectorized inner–product computation. CMSSL_DNRM2, in addition, requires a square root evaluation which is equivalent to 14 floating–point operations on the CM–200. The peak efficiencies for CMSSL_SNRM2, CMSSL_SDOT, and CMSSL_SAXPY are 70%, 75%, and 90%, respectively. The higher efficiency for operations in 64–bit precision is due to the fact that the data paths to memory are 32–bits wide, and thus, memory operations require twice the time, thereby reducing the relative importance of looping overhead, pipeline lengths, etc. Figures 3, 4 and 5 show the aggregate floating–point rates for a 2048 processor CM–200.

Both the _DOT and _AXPY routines first load one operand into the register file, then perform the required operations while reading the second operand from memory. The maximum length of a single vector operation is limited by the size of the register file. With 32 registers, our implementation uses a maximum vector length of 29 for the _DOT routines and a maximum of 30 for the _AXPY routines (the remaining registers being used for other variables). The pipeline length of the adder is two. For the _DOT routines two partial results are computed by using the output of the adder as one of the inputs, and the output from the multiplier as the other input. The two partial results (corresponding to the depth of the adder) are added for the final result after the vectorized accumulation is completed.

The performance for both the _DOT and _AXPY computation increases with the vector length, but a penalty is incurred each time a new vector must be loaded into the register file. The influence of the vector length and the penalty for loading vectors into registers, is easily seen in Figures 4 and 5.

For the _NRM2 computation there is no need to preload a vector into the register file. It is possible to apply the value read from memory to both inputs of the multiplier, hence directly squaring the value read from memory. By accumulating the squared values as they are computed, there is, in effect, no upper limit on the vector length. However, because of limitations in the Connection Machine Instruction Set (CMIS), the vector length is limited to a maximum of 29, just as in the _DOT routines. The effect of the limited vector length is clearly visible in Figure 3.

The difference in performance between the DNRM2, DDOT, and DAXPY routines is apparent from Figure 6. The difference is mostly due to the difference in the need for memory accesses. The peak performance, $R_\infty$, and the array size for half of peak predicted performance based on memory references, $n_{\frac{1}{2}}$, are summarized in Table 3. The Table also states the maximum vector length in a vector instruction. Note that the stronger the dependence of the peak performance on the memory bandwidth, the shorter the vector length for half of peak performance.

# 5 Level–2 LBLAS

Of the level–2 BLAS, we have implemented LBLAS versions of matrix–vector multiplication (_GEMV) and outer–products (_GER). As in the BLAS, one routine is used for both matrix–vector and vector–matrix multiplication. One or the other function is obtained by an appropriate specification of strides for the matrix axes. The basic operation in our CM–200 CMSSL__GEMV routines is the vector operation $z \leftarrow \alpha x + z$. The accumulation vector $z$ and the coefficient $\alpha$ are kept in registers, as discussed in Section 1. The CMSSL__AXPY routines are *not* used (since $z$ is explicitly kept in registers). In the LBLAS for the CM–5, additional loop orderings are used, as will be described elsewhere. While _AXPY routines are not used for matrix–vector multiplication in the LBLAS, they are used for the local outer–product routines. The number of floating–point operations per memory reference for matrix–vector (and vector–matrix) multiplication is twice that

| V-length | SNRM2 | DNRM2 | SDOT | DDOT | SAXPY | DAXPY |
|---|---|---|---|---|---|---|
| 1 | 0.21 | 0.16 | 0.21 | 0.16 | 0.21 | 0.16 |
| 2 | 0.45 | 0.47 | 0.35 | 0.35 | 0.41 | 0.29 |
| 3 | 0.67 | 0.68 | 0.52 | 0.51 | 0.58 | 0.42 |
| 4 | 0.89 | 0.89 | 0.68 | 0.66 | 0.80 | 0.54 |
| 5 | 1.10 | 1.09 | 0.83 | 0.79 | 0.97 | 0.65 |
| 8 | 1.70 | 1.63 | 1.27 | 1.15 | 1.40 | 0.95 |
| 10 | 2.08 | 1.95 | 1.53 | 1.36 | 1.65 | 1.13 |
| 15 | 2.96 | 2.66 | 2.13 | 1.79 | 2.15 | 1.41 |
| 20 | 3.13 | 3.26 | 2.65 | 2.13 | 2.55 | 1.61 |
| 25 | 4.48 | 3.76 | 3.10 | 2.40 | 2.86 | 1.76 |
| 30 | 4.34 | 3.64 | 3.03 | 2.21 | 3.11 | 1.87 |
| 40 | 5.42 | 4.29 | 3.71 | 2.70 | 3.07 | 1.89 |
| 50 | 6.34 | 4.83 | 4.23 | 2.96 | 3.37 | 2.02 |
| 60 | 6.65 | 5.00 | 4.41 | 3.02 | 3.53 | 2.09 |
| 70 | 7.30 | 5.38 | 4.75 | 3.19 | 3.56 | 2.11 |
| 80 | 7.92 | 5.69 | 5.06 | 3.33 | 3.71 | 2.18 |
| 100 | 8.51 | 5.98 | 5.32 | 3.45 | 3.77 | 2.22 |
| 120 | 8.96 | 6.19 | 5.51 | 3.53 | 3.93 | 2.28 |
| 140 | 9.71 | 6.53 | 5.86 | 3.68 | 4.00 | 2.31 |
| 256 | 11.20 | 7.15 | 6.52 | 3.92 | 4.20 | 2.39 |
| 512 | 12.30 | 7.57 | 6.97 | 4.07 | 4.32 | 2.43 |
| 1024 | 13.00 | 7.80 | 7.24 | 4.15 | 4.42 | 2.46 |
| 2048 | 13.40 | 7.94 | 7.38 | 4.20 | 4.46 | 2.48 |
| 4096 | 13.60 | 8.00 | 7.45 | 4.22 | 4.49 | 2.49 |
| 8192 | 13.70 | 8.04 | 7.49 | 4.24 | 4.50 | 2.49 |
| Est. peak | 20.00 | 10.00 | 10.00 | 5.00 | 5.00 | 2.50 |

Table 2: Level–1 LBLAS floating–point rates in Mflop/s on *each* CM–200 floating–point processor.

| | DNRM2 | DDOT | DAXPY |
|---|---|---|---|
| $R_\infty$ | 8.04 | 4.24 | 2.49 |
| $n_{\frac{1}{2}}$ | 35 | 20 | 10 |
| Max V-length | 29 | 29 | 30 |

Table 3: Peak performance and vector length for half of peak performance in 64–bit precision for CMSSL level–1 LBLAS on *each* node of a CM–200 node.

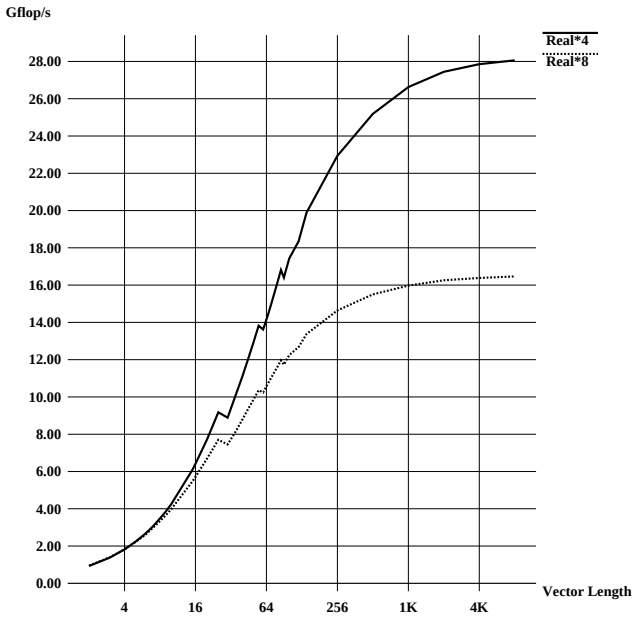


Figure 3: The aggregate floating–point rate in Gflop/s for the local CMSSL_SNRM2 and CMSSL_DNRM2 routines on a 2048 processor CM–200.

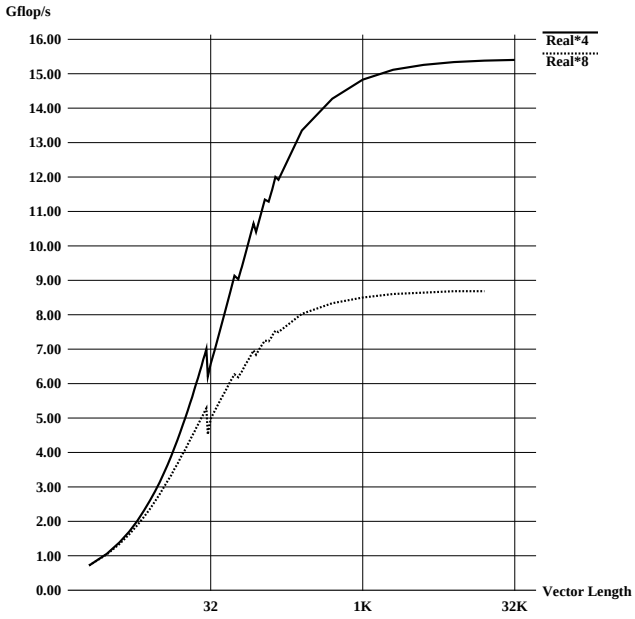


Figure 4: The aggregate floating–point rate in Gflop/s for the local CMSSL_SDOT and CMSSL_DDOT routines on a 2048 processor CM–200.

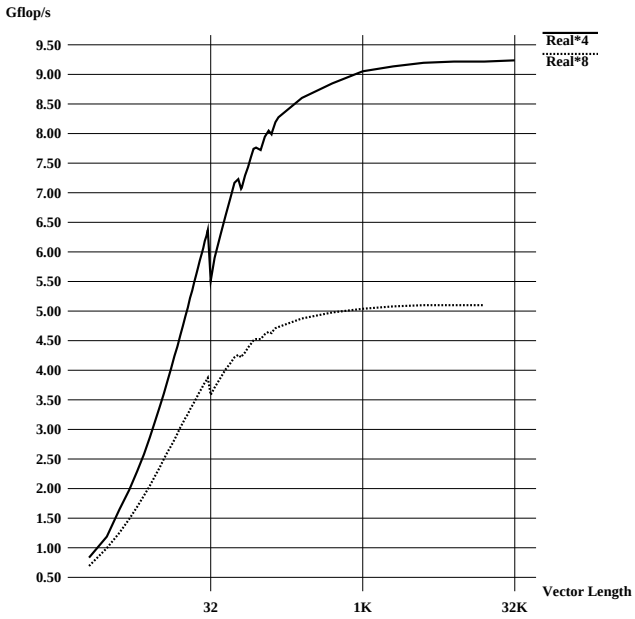


Figure 5: The aggregate floating–point rate in Gflop/s for the local CMSSL_SAXPY and CMSSL_DAXPY routines on a 2048 processor CM–200.

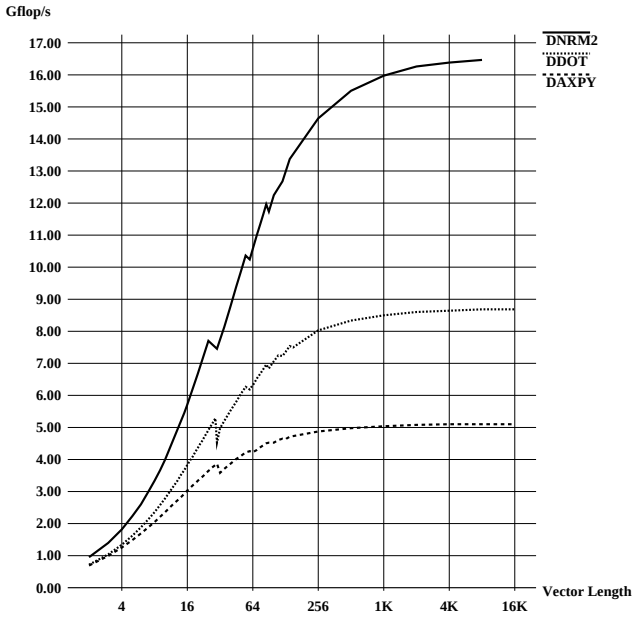


Figure 6: The aggregate floating-point rates in Gflop/s for the local CMSSL_DNRM2, CMSSL_DDOT and CMSSL_DAXPY routines on a 2048 processor CM–200.

of an outer–product. With the CM–200 processor architecture, the expected peak performance of matrix–vector multiplication is not twice, but, in fact, three or four times that of the outer–product computation, as explained below.

## 5.1 Estimated peak performance

For the matrix–vector multiplication $y \leftarrow Ax + y$, two floating–point operations are required for each matrix element. The minimum number of loads from memory is $P+Q+PQ$ for a matrix of shape $P \times Q$, and the minimum number of stores is $P$. For $PQ \gg P, Q$, the peak processing rate approaches two floating–point operations per memory load/store operation, or, for the CM–200, 20 Mflop/s per processor in 32–bit precision and 10 Mflop/s per processor in 64–bit precision. Hence, matrix–vector multiplication can achieve close to the peak arithmetic performance in 32–bit precision.

For the outer–product computation $C \leftarrow xy^T + C$, the number of elements that must be loaded from memory is $P+Q+PQ$, i.e., the same as for matrix–vector multiplication. But, $PQ$ elements must be stored. With each store in 32–bit precision requiring two cycles, the peak rate approaches two floating–point operations for every three cycles, instead of two operations for every cycle in matrix–vector multiplication. Thus, for $PQ \gg P, Q$, the peak estimated performance for the outer–product $C \leftarrow xy^T + C$ is only $\frac{1}{3}$rd of the performance for the operation $y \leftarrow Ax + y$. The asymptotic peak performance for outer–products can be achieved by preloading $x$ and a (few) component(s) of $y$ into the register file, then reading a column of $C$ from memory while performing the operation $z(:) \leftarrow y(i)x(:) + C(:, i)$ with the destination of $z(:)$ being a vector register. The desired result is obtained through the store operation $C(:, i) \leftarrow z$. Two vector registers are required: one for $x$, one for $z$.

For the outer–product operation $C \leftarrow xy^T$, only two vectors must be loaded from memory. But, there is only one floating–point operation to be performed for each element of $C$. With stores requiring two cycles, the performance for $PQ \gg P, Q$ approaches one quarter of the peak performance for matrix–vector multiplication.

Because of the limited number of registers in each CM–200 processor, and the pipeline start–up and shut–down times, we choose to use the CMSSL__AXPY routine for our outer–product routine. Thus, in the CM–200 LBLAS, the peak floating–point performance per processor in 32–bit precision is 5 Mflop/s, and 2.5 Mflop/s in 64–bit precision for both $C \leftarrow xy^T + C$ and $C \leftarrow xy^T$. Hence, the peak outer–product performance on the CM–200 is only a quarter of the peak performance of the CMSSL__GEMV routines. The estimated peak arithmetic performances for matrix–vector multiplication and outer–product computations are summarized in Table 4.

## 5.2 Matrix–vector multiplication

In the CMSSL__GEMV routine the accumulation vector $z$ in $z(:) \leftarrow x(i)A(:, i) + z(:)$ is allocated in the register file. Elements of the vector $x$ are preloaded into the register file,

| Function | SGER | DGER | SGEMV | DGEMV |
|---|---|---|---|---|
| Est. peak. perf. | 5 | 2.5 | 20 | 10 |

Table 4: Estimated peak floating–point rates in Mflop/s for the CMSSL_GER and CMSSL__GEMV routines on *each* CM–200 processor.

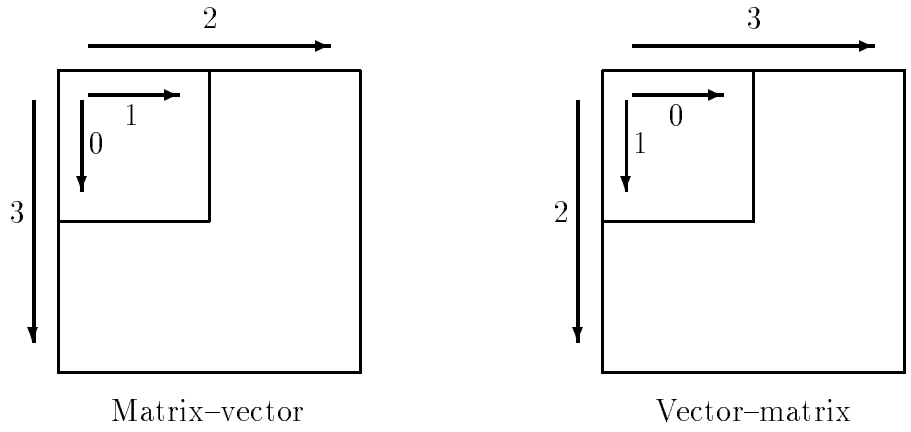


Figure 7: The CM–200 LBLAS loop ordering for matrix–vector and vector–matrix multiplication.

while the columns $A(:, i)$ are read from memory as needed. With 32 registers, the looping over the rows and the columns of the matrix must be divided, in general, into two loops each. The loop orderings used for matrix–vector multiplication in the CM–200 LBLAS are shown in Figure 7. The innermost, vectorized loop is labeled 0. The outermost loop is labeled 3.

Several vector instructions corresponding to successive iterations of loop 1 can be *chained* together as a macro vector operation. The chaining of individual vector instructions results in one pipeline start–up and shut–down for the entire chain of vector instructions, instead of a pipeline start–up and shut–down for each vector instruction. To allow for vector operations to be chained, the necessary updating of pointers and base addresses must be performed concurrently with the execution of a vector instruction. On the CM–200, this form of concurrency can only be achieved in microcode. Moreover, the number of vector instructions chained together cannot be a run–time argument. The performance gain from chaining is typically in the 5 – 10 % range.

The accumulation vector $z$ and the coefficient vector $x$ must share the register set in a processor. With $v_1$ registers allocated to $z$ and $v_2$ registers to $x$, the number of vector loads of $x$ is $\frac{Q}{v_2}\frac{P}{v_1}$, for the loop order shown in Figure 7. $x$ is loaded $\frac{P}{v_1}$ times. The total number of stores of $v_1$ elements each is $\frac{P}{v_1}$. Clearly, with respect to load and store operations, it is desirable to maximize $v_1$. In determining the optimal values of $v_1$ and $v_2$, the overheads for vector loads, for the chained operations, and for the vector stores must also be included. In the LBLAS for the CM–200, $v_1 = 22$ and $v_2 = 8$. Two of the

32 registers in a processor are used for temporary variables.

The timings reported in Tables 5 and 6 are based on chaining sets of 8 vector instructions, when possible. A matrix of arbitrary shape is divided into the maximum number of sets of 22 rows and 8 columns. The remaining columns are treated by independent vector instructions. Remaining rows are treated in a manner analogous to a set of 22 rows, except for the difference in vector length. All submatrices with 8 columns are handled by chained vector instructions. The effects of these restrictions are apparent in Figures 8 and 9, which show the performance of our matrix–vector multiplication routines. Each iteration in loop 2 requires a load of a new segment of the vector $x$ as well as a new chained vector operation. Each iteration in loop 3, in addition, requires a segment of the vector $y$ to be stored, and a new segment of the vector $y$ to be loaded from memory for the operation $y \leftarrow Ax + y$ on a new set of rows. The peak efficiency in 32–bit precision is 90% and in 64–bit precision, 92%.

The vector length (number of iterations in loop 0) has a very strong influence on performance up to the maximum vector length of 22. For four columns, the performance increases by a factor of about 5.1 when the vector length increases from 2 to 22, while for 500 columns, the increase in performance is approximately a factor of 1.8 for the same range of vector lengths. The overhead associated with loop 2, i.e., successive chained vector operations, is apparent in Figure 10.

For the timings in Figures 8 and 9, the stride for the elements in the inner loop is 1, while the stride for loop 1 is $P$. With a DRAM page size of 1 kbyte, and arrays aligned with DRAM page boundaries, no page fault occurs in the inner loop for $P \leq 256$ in 32–bit precision, or $P \leq 128$ in 64–bit precision. With arrays not aligned with DRAM pages, a page fault may occur at most for one instance of loop 0 for these values of $P$. One additional DRAM page fault will be encountered in loop 0 for every 256 or 128 rows, respectively, added to the matrix. For sufficiently small arrays aligned with DRAM pages, no page faults occur in loop 1. However, for $P > 256$ in 32–bit precision, or $P > 128$ in 64–bit precision, a page fault occurs for every iteration in every instance of loop 1. The relatively larger impact of the DRAM page faults on performance is clear by comparing Figures 8 and 9. With one page fault per iteration in loop 1 and no page faults in loop 0, the number of cycles per iteration in loop 1 increases from 22 to 23 in 32–bit precision, while it increases from 44 to 45 in 64–bit precision.

Interchanging the order of loops 0 and 1, as required for vector–matrix multiplication using the routine CMSSL_GEMV on the CM–200, may significantly increase the number of page faults for a given matrix layout in memory. With the matrix $B$ being of shape $Q \times R$ for the operation $y^T \leftarrow x^T B + y^T$, the stride for the inner loop increases from 1 to $Q$ by interchanging the ordering of loops 0 and 1. For $Q > 256$ in 32–bit precision, one page fault occurs for every memory access. For $Q > 128$ in 64–bit precision, one page fault occurs for every two memory cycles, since a 64–bit word is stored as two contiguous 32–bit words. Thus, for $Q = 128$ the peak performance of vector–matrix multiplication is $\frac{2}{3}$rds of the corresponding matrix–vector multiplication performance for the same matrix layout, in both 32–bit and 64–bit precision. This behavior is apparent in Figures 11 and 12. Tables 8 and 9 contain performance data for vector–matrix multiplication, i.e., for the

| CMSSL_SGEMV ($y \leftarrow Ax + y$) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of | Number of columns | | | | | | | | | | | |
| Rows | 2 | 3 | 4 | 5 | 8 | 16 | 24 | 32 | 64 | 128 | 512 | 2048 |
| 2 | 0.84 | 1.17 | 1.46 | 1.71 | 2.91 | 4.62 | 5.74 | 6.53 | 7.83 | 8.92 | 9.99 | 10.30 |
| 3 | 1.18 | 1.64 | 2.03 | 2.38 | 3.93 | 6.04 | 7.35 | 8.21 | 9.80 | 10.80 | 11.90 | 12.20 |
| 4 | 1.62 | 2.23 | 2.73 | 3.17 | 5.11 | 7.51 | 5.17 | 9.53 | 11.20 | 12.30 | 13.20 | 13.50 |
| 5 | 1.93 | 2.63 | 3.22 | 3.72 | 5.86 | 8.43 | 9.72 | 10.60 | 12.20 | 13.20 | 14.10 | 14.40 |
| 8 | 2.73 | 3.68 | 4.45 | 5.09 | 7.61 | 10.20 | 11.70 | 12.50 | 14.00 | 14.90 | 15.70 | 16.00 |
| 16 | 4.16 | 5.47 | 6.50 | 7.32 | 9.99 | 12.80 | 14.00 | 14.80 | 16.10 | 16.80 | 17.40 | 17.60 |
| 22 | 4.61 | 6.53 | 7.43 | 8.28 | 11.00 | 13.70 | 14.90 | 15.60 | 16.70 | 17.40 | 17.90 | 18.00 |
| 32 | 4.60 | 5.96 | 6.73 | 7.56 | 10.60 | 13.20 | 14.40 | 15.10 | 16.30 | 16.90 | 17.40 | 17.60 |
| 44 | 5.17 | 6.49 | 9.66 | 8.48 | 11.40 | 14.00 | 15.10 | 15.70 | 16.80 | 17.40 | 17.90 | 18.00 |
| 64 | 5.35 | 6.85 | 7.85 | 8.72 | 11.70 | 14.20 | 15.30 | 15.90 | 16.90 | 17.50 | 17.90 | — |
| 128 | 5.63 | 7.09 | 8.14 | 8.94 | 12.10 | 14.50 | 15.50 | 16.10 | 17.00 | 17.50 | 17.90 | — |
| 512 | 5.84 | 7.30 | 8.36 | 9.15 | 12.20 | 14.40 | 15.40 | 15.90 | 16.80 | 17.30 | — | — |
| 2048 | 5.93 | 7.40 | 8.46 | 9.25 | 12.10 | 14.30 | 15.20 | 15.70 | — | — | — | — |

Table 5: The floating–point rates in Mflop/s achieved in *each* CM–200 processor for matrix–vector multiplication.

CMSSL_GEMV routine with the stride in the inner loop equal to the length of the row axis of the matrix. The performance losses due to page faults are summarized in Table 7.

With unit stride in loop 0, the impact on performance of DRAM page faults only amounts to a few percent. With unit stride in loop 1, and a stride in loop 0 equal to the length of the axis for loop 1, the maximum estimated performance loss in 32–bit precision is 50%, and in 64–bit precision 33.3%. The measured peak performance with the maximum number of page faults is 62% of peak measured performance in 32–bit precision, and 77% of peak measured performance in 64–bit precision, for a fixed loop order. However, a higher peak performance is attained if loop 0 is given unit stride, and loop 1 a stride equal to the length of the axis for loop 0. Measuring the performance loss under page faults relative to this loop ordering yields a loss of 42% (instead of 38%) in 32–bit precision, and a loss of 28% (instead of 23%) in 64–bit precision. Because of the page faults with loop 1 having unit stride, the performance for this loop order never reaches the level achieved with loop 0 having unit stride. Though there is a difference in performance for large matrices depending upon which of the loops 0 and 1 have unit stride, the performance is essentially independent of the stride for loops 0 and 1 for small matrices.

In the CMSSL, any axis of a multidimensional array can be chosen as the row axis and as the column axis. Thus, whether a row– or column–oriented array layout is used, either the row or the column axis for a matrix may have the smaller stride of the two, and none of the selected axes may have unit stride. Since it is desirable to avoid as many page faults as possible in the inner loop, choosing loop 0 to enumerate elements along the axis with the smaller stride of the two matrix axes is a plausible strategy. However, this strategy may change the character of the algorithm from being _AXPY–like to being _DOT–like. Due to the inefficiency on the CM–200 of a single inner product of length at most equal to the maximum vector length, it may be desirable to maintain the ordering

| CMSSL_DGEMV ($y \leftarrow Ax + y$) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of | Number of columns | | | | | | | | | | | |
| Rows | 2 | 3 | 4 | 5 | 8 | 16 | 24 | 32 | 64 | 128 | 512 | 2048 |
| 2 | 0.76 | 1.04 | 1.28 | 1.48 | 2.33 | 3.36 | 3.88 | 4.19 | 4.85 | 5.28 | 5.65 | 5.75 |
| 3 | 1.07 | 1.45 | 1.76 | 2.02 | 3.06 | 4.20 | 4.77 | 5.15 | 5.80 | 6.21 | 6.57 | 6.66 |
| 4 | 1.32 | 1.77 | 2.14 | 2.44 | 3.58 | 4.76 | 3.17 | 5.76 | 6.41 | 6.81 | 7.15 | 7.24 |
| 5 | 1.53 | 2.04 | 2.45 | 2.78 | 4.00 | 5.23 | 5.83 | 6.21 | 6.84 | 7.23 | 7.54 | 7.63 |
| 8 | 2.02 | 2.66 | 3.15 | 3.54 | 4.80 | 6.09 | 6.69 | 7.04 | 7.62 | 7.96 | 8.23 | 8.31 |
| 16 | 2.76 | 3.54 | 4.03 | 4.49 | 5.84 | 7.08 | 7.62 | 7.92 | 8.42 | 8.69 | 8.92 | 8.97 |
| 22 | 3.31 | 4.19 | 4.45 | 4.91 | 6.19 | 7.40 | 7.91 | 8.19 | 8.65 | 8.91 | 9.11 | 9.17 |
| 32 | 2.91 | 3.71 | 4.24 | 4.69 | 6.06 | 7.24 | 7.74 | 8.02 | 8.47 | 8.72 | 8.92 | — |
| 44 | 3.17 | 3.99 | 4.59 | 5.03 | 6.33 | 7.50 | 7.98 | 8.25 | 8.68 | 8.92 | 9.11 | — |
| 64 | 3.27 | 4.10 | 4.68 | 5.12 | 6.45 | 7.57 | 8.04 | 8.29 | 8.71 | 8.94 | 9.11 | — |
| 128 | 3.38 | 4.20 | 4.79 | 5.23 | 6.54 | 7.62 | 8.07 | 8.32 | 8.71 | 8.93 | — | — |
| 512 | 3.46 | 4.29 | 4.87 | 5.30 | 6.54 | 7.57 | 8.00 | 8.23 | 8.60 | — | — | — |
| 2048 | 3.49 | 4.33 | 4.91 | 5.34 | 6.57 | 7.60 | 8.02 | — | — | — | — | — |

Table 6: The floating–point rates in Mflop/s achieved in *each* CM–200 processor for matrix–vector multiplication.

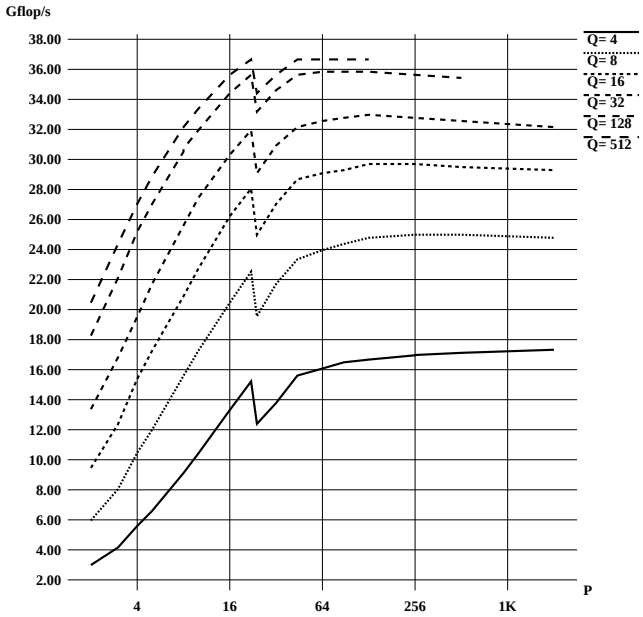


Figure 8: The aggregate performance for matrix–vector multiplication in 32–bit precision on a 2048 processor CM–200. The matrix shape is $P \times Q$.

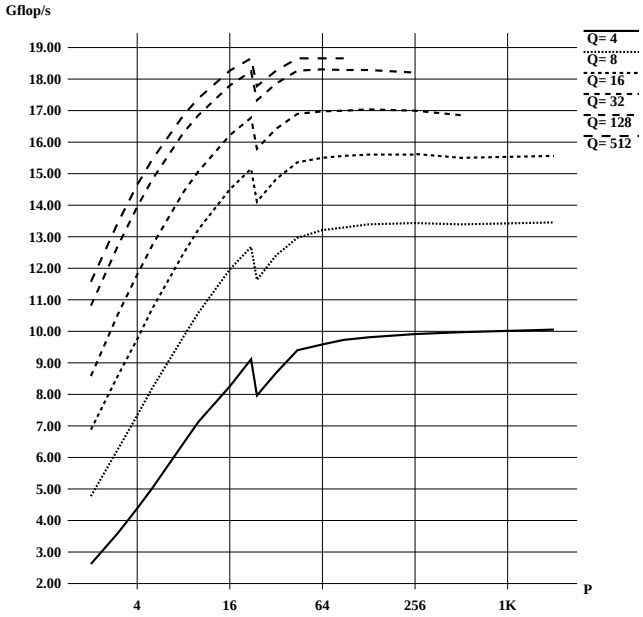


Figure 9: The aggregate performance for matrix–vector multiplication in 64–bit precision on a 2048 processor CM–200. The matrix shape is $P \times Q$.

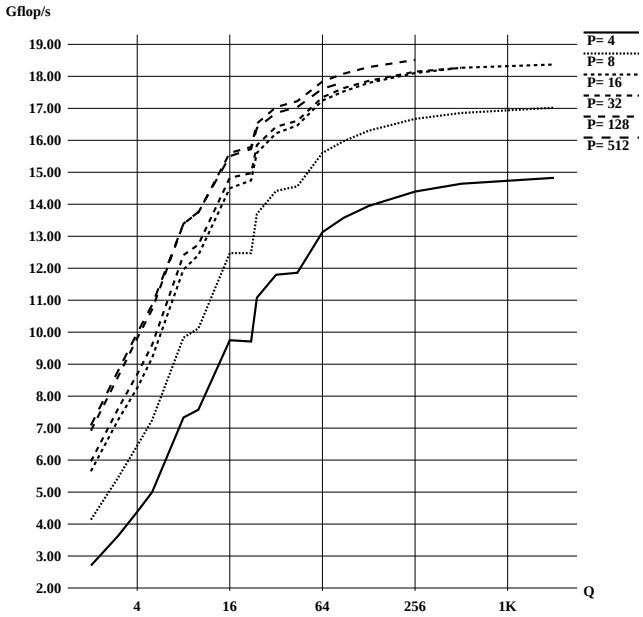


Figure 10: The overhead associated with successive chained vector operations is mainly the cause of the irregularities in the curves in this Figure. 64–bit precision.

| Precision | Fixed loop ordering | Best loop ordering |
|---|---|---|
| 32–bit | 38% | 42% |
| 64–bit | 23% | 28% |

Table 7: Possible peak performance loss due to DRAM page faults with the axis for loop 1 having unit stride, and the axis for loop 0 having a stride equal to the length of the axis for loop 1.

| CMSSL_SGEMV with unit stride in loop 1 ($y^T \leftarrow x^T A + b^T$) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of Rows | Number of columns | | | | | | | | | | | |
| | 2 | 3 | 4 | 5 | 8 | 16 | 24 | 32 | 64 | 128 | 512 | 2048 |
| 2 | 0.84 | 1.18 | 1.62 | 1.93 | 2.73 | 4.16 | 3.88 | 4.60 | 5.42 | 5.65 | 5.84 | 5.92 |
| 3 | 1.17 | 1.64 | 2.23 | 2.63 | 3.68 | 5.47 | 5.10 | 5.96 | 6.87 | 7.11 | 7.30 | 7.40 |
| 4 | 1.46 | 2.03 | 2.73 | 3.22 | 4.45 | 6.50 | 6.47 | 6.94 | 7.92 | 8.16 | 8.35 | 8.45 |
| 5 | 1.71 | 2.38 | 3.17 | 3.72 | 5.09 | 7.32 | 6.75 | 7.71 | 8.73 | 8.95 | 9.13 | 9.24 |
| 8 | 2.91 | 3.93 | 5.11 | 5.86 | 7.60 | 9.92 | 9.53 | 10.50 | 11.60 | 11.90 | 12.20 | 12.30 |
| 16 | 4.62 | 6.04 | 7.51 | 8.43 | 10.30 | 12.50 | 12.10 | 13.00 | 13.90 | 14.10 | 14.40 | 14.40 |
| 22 | 4.84 | 6.07 | 7.49 | 8.24 | 10.10 | 12.60 | 11.80 | 12.90 | 13.80 | 14.00 | 14.10 | 14.20 |
| 32 | 6.53 | 8.21 | 9.70 | 10.40 | 12.50 | 14.20 | 13.80 | 14.50 | 15.20 | 15.50 | 15.50 | 15.50 |
| 44 | 6.47 | 8.20 | 9.52 | 10.40 | 12.00 | 14.20 | 13.50 | 14.40 | 15.10 | 15.10 | 15.30 | 15.30 |
| 64 | 8.00 | 9.43 | 11.30 | 11.60 | 14.10 | 14.70 | 14.60 | 14.80 | 15.50 | 15.70 | 15.70 | — |
| 128 | 8.92 | 9.99 | 12.30 | 12.20 | 14.90 | 15.30 | 14.80 | 15.00 | 15.40 | 15.40 | 15.40 | — |
| 512 | 9.99 | 8.65 | 10.00 | 10.00 | 11.40 | 12.10 | 12.10 | 12.10 | 12.30 | 12.30 | — | — |
| 2048 | 6.94 | 7.69 | 8.14 | 8.43 | 8.91 | 9.35 | 9.20 | 9.35 | — | — | — | — |

Table 8: The floating–point rates in Mflop/s achieved in *each* CM–200 processor for vector–matrix multiplication.

of loops 2 and 3. This allows the final, nonvectorized reduction to be deferred until the end, and to be performed once for each row of the matrix. The expense on the CM–200 is that two registers are required for each matrix row in loop 1. Because of the limited number of registers in each CM–200 floating–point processor, and the relative inefficiency of the inner product instruction, a performance gain would occur only in relatively few cases. However, on the CM–5, where each vector unit has 64 64–bit registers (128 32–bit registers), and a page fault amounts to 2.5 cycles, we have also implemented the _DOT–like matrix–vector multiplication loop ordering. The CM–5 implementation will be described elsewhere.

## 5.3 Outer–products

The outer–product computation $C \leftarrow xy^T + C$ is performed using the _AXPY routines. The _AXPY operations may be row– or column–oriented as shown in Figure 13. In the column–oriented scheme, a column $C(:, i)$ is loaded into the register file, together with

| CMSSL_DGEMV with unit stride in loop 1 ($y^T \leftarrow x^T A + b^T$) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of | Number of columns | | | | | | | | | | | |
| Rows | 2 | 3 | 4 | 5 | 8 | 16 | 24 | 32 | 64 | 128 | 512 | 2048 |
| 2 | 0.76 | 1.07 | 1.32 | 1.53 | 2.02 | 2.76 | 2.65 | 2.94 | 3.28 | 3.38 | 3.46 | 3.49 |
| 3 | 1.04 | 1.45 | 1.77 | 2.04 | 2.66 | 3.54 | 3.37 | 3.71 | 4.10 | 4.20 | 4.28 | 4.32 |
| 4 | 1.28 | 1.76 | 2.14 | 2.45 | 3.15 | 8.60 | 4.12 | 4.27 | 4.69 | 4.78 | 4.86 | 4.90 |
| 5 | 1.48 | 2.02 | 2.44 | 2.78 | 3.54 | 4.53 | 4.30 | 4.69 | 5.12 | 5.22 | 5.29 | 5.33 |
| 8 | 2.33 | 3.06 | 3.58 | 4.00 | 4.80 | 5.78 | 5.65 | 5.99 | 6.38 | 6.48 | 6.57 | 6.60 |
| 16 | 3.36 | 4.20 | 4.79 | 5.18 | 6.10 | 6.93 | 6.80 | 7.08 | 7.40 | 7.51 | 7.54 | 7.57 |
| 22 | 3.06 | 4.15 | 4.72 | 5.15 | 6.00 | 7.04 | 6.76 | 7.13 | 7.49 | 7.52 | 7.57 | 7.60 |
| 32 | 4.23 | 5.05 | 5.77 | 6.03 | 7.04 | 7.56 | 7.48 | 7.65 | 7.95 | 8.01 | 8.04 | — |
| 44 | 4.12 | 5.08 | 5.61 | 5.98 | 6.73 | 7.63 | 7.46 | 7.71 | 8.00 | 8.01 | 8.05 | — |
| 64 | 4.85 | 5.54 | 6.41 | 6.43 | 7.62 | 8.02 | 7.81 | 7.96 | 8.17 | 8.19 | 8.20 | — |
| 128 | 5.28 | 5.65 | 6.81 | 6.57 | 7.27 | 7.87 | 7.66 | 7.78 | 8.00 | 7.98 | — | — |
| 512 | 4.46 | 4.97 | 5.31 | 5.51 | 5.85 | 6.18 | 6.07 | 6.18 | 6.27 | — | — | — |
| 2048 | 4.52 | 5.03 | 5.36 | 5.55 | 5.89 | 6.20 | 6.10 | — | — | — | — | — |

Table 9: The floating–point rates in Mflop/s achieved in *each* Connection Machine system CM-200 processor for vector–matrix multiplication.

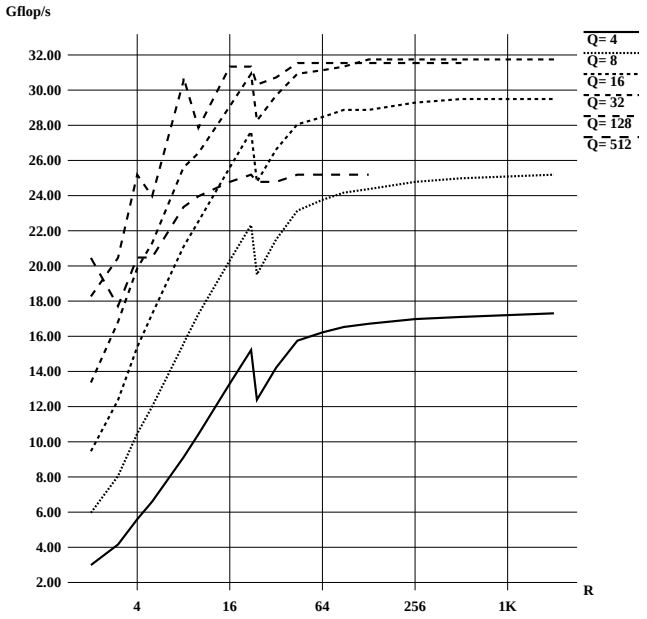


Figure 11: The aggregate performance for vector–matrix multiplication in 32–bit precision on a 2048 processor CM–200. The matrix shape is $Q \times R$.

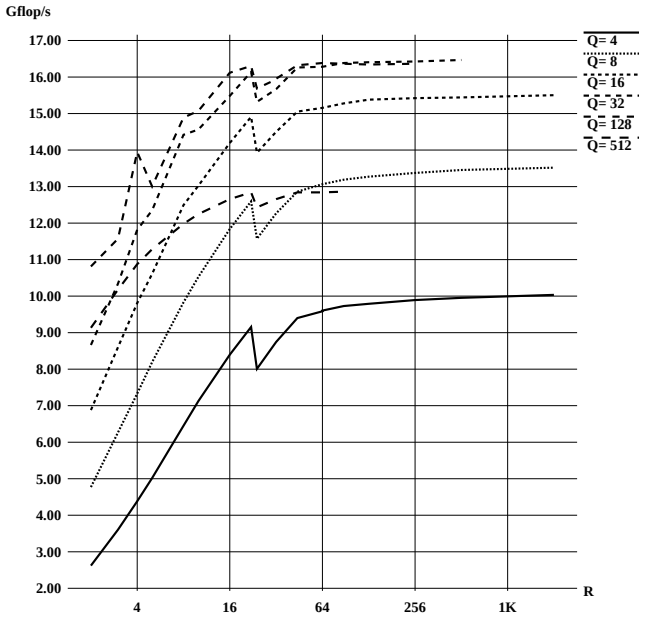


Figure 12: The aggregate performance for vector–matrix multiplication in 64–bit precision on a 2048 processor CM–200. The matrix shape is $Q \times R$.

$y(i)$. Then, the operation $z(:) \leftarrow y(i)x(:) + C(:,i)$ is performed with $x$ read from memory, and $z$ overwriting $C(:,i)$ in the register file, followed by a store, $C(:,i) \leftarrow z(:)$. With 32 registers, the columns are partitioned into segments of length 30 in our implementation. The index space of the matrix is traversed as shown in Figure 13.

The choice between a row– or column–oriented algorithm is based on the strides within rows and columns of $C$, $x$, and $y$, and the shape of $C$. For a row–oriented algorithm it suffices to consider $y$ and one row of $C$, while for a column–oriented algorithm $x$ and one column of $C$ is considered. The loop order is determined as described in the next section for matrix–matrix multiplication.

Performance data for rank–1 updates on square matrices are given in Table 10 and Figure 14. The performance data are almost identical to those of the CMSSL__AXPY routines in Table 2, as expected. The peak efficiency for 32–bit precision is about 85%, while the peak for 64–bit precision approaches 100%. The latter figure shows that our estimate of a store requiring two cycles is too conservative. The influence of the vector length is apparent in Figure 14.

# 6 Matrix–matrix multiplication

Local matrix–matrix multiplication in the CM–200 LBLAS is based on the level–1 or level–2 LBLAS. Which routine is called depends entirely upon matrix shape, as does the

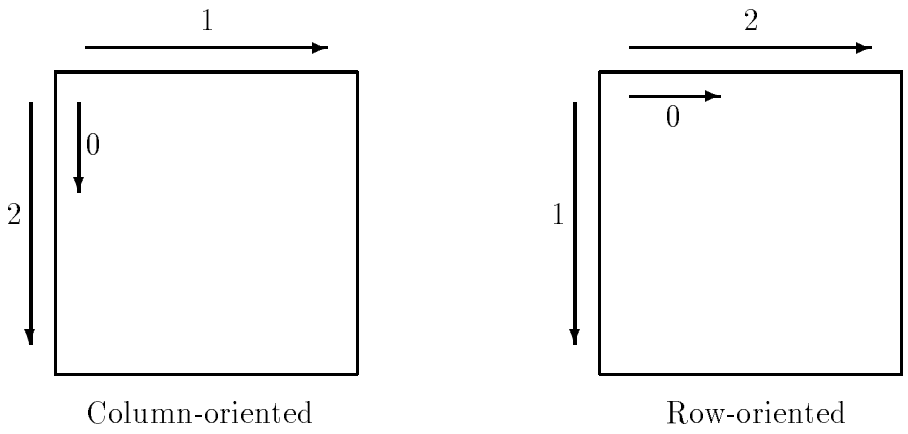


Figure 13: Loop ordering for outer–product computation.

| No. of rows & columns | SGER | DGER |
|---|---|---|
| 2 | 0.41 | 0.34 |
| 4 | 0.80 | 0.61 |
| 8 | 1.40 | 1.00 |
| 10 | 1.65 | 1.14 |
| 16 | 2.21 | 1.45 |
| 22 | 2.63 | 1.67 |
| 28 | 2.96 | 1.83 |
| 32 | 2.63 | 1.73 |
| 40 | 3.01 | 1.88 |
| 46 | 3.20 | 1.96 |
| 52 | 3.36 | 2.03 |
| 58 | 3.51 | 2.09 |
| 64 | 3.39 | 2.05 |
| 128 | 3.90 | 2.26 |
| 256 | 4.20 | 2.35 |
| 512 | 4.29 | — |
| Est. peak | 5.00 | 2.50 |

Table 10: The floating–point rate in Mflop/s achieved in *each* CM–200 processor for rank–1 updates of square matrices.

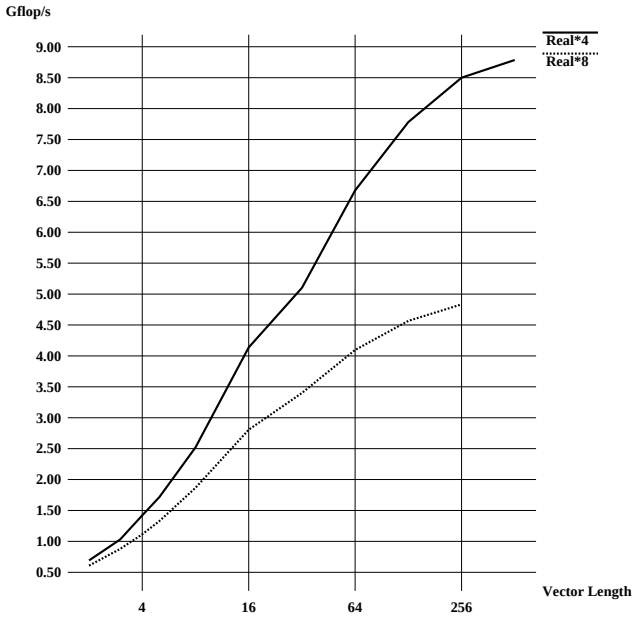


Figure 14: The aggregate performance for rank–1 updates of square matrices on a 2048 processor CM–200.

calling sequence when level–2 LBLAS are called. How the calling sequence is determined is addressed in this section. A CMSSL_DOT routine is only called when $P = R = 1$, and $Q > 1$. (Recall that in the matrix multiplication $C \leftarrow A \times B$, $A$ is of shape $P \times Q$ and $B$ is of shape $Q \times R$). A CMSSL_AXPY routine is chosen only when $P > 1$, and $Q = R = 1$ or $P = Q = 1$ and $R > 1$. A CMSSL_GER routine is chosen if $Q = 1$ and $P, R > 1$. For all other cases the matrix–matrix multiplication is performed through calls to a CMSSL_GEMV routine. Making the loop on the $P$–axis or the $R$–axis the innermost loop and the other loop the outermost loop is a choice made as a trade–off between overhead, DRAM page faults, and pipeline lengths. Thus, the loop order is a function of the shapes of the operands and the strides along the row and column axes, i.e., the data layout in each node, and must be determined at run–time.

If at least one of the operands is a true matrix, i.e., both of its axes have a length greater than 1, then it is clear from Tables 2, 5, 6, 8, and 9 that, for the LBLAS, a rank–1 routine (effectively an _AXPY routine) is never competitive with a CMSSL_GEMV routine. From the same set of tables it is also clear that the CMSSL_GEMV routines offer superior performance compared to the CMSSL_DOT routines, for most matrix shapes. Indeed, there is only one case (for which we have performance data) in which the CMSSL_SDOT routine performs better than the CMSSL_SGEMV routine, namely for vector–matrix multiplication with a matrix of shape $2048 \times 2$ layed out in column major order. In this case, the inner loop is of length two with a stride of 2048. The performance of the CMSSL_SDOT routine is 7.38 Mflop/s per processor, compared to 6.94 Mflop/s per processor for the CMSSL_SGEMV routine. The reason for the lower

| $P = Q = R$ | 4 | 10 | 16 | 22 | 28 | 34 | 40 | 46 | 52 | 58 | 64 | 70 | 76 | 88 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SGEMM | 2.33 | 7.83 | 12.10 | 13.40 | 13.40 | 14.70 | 15.80 | 14.90 | 15.70 | 16.30 | 16.80 | 16.20 | 16.60 | 17.20 |
| DGEMM | 1.89 | 5.03 | 6.88 | 7.44 | 7.45 | 7.94 | 8.32 | 8.04 | 8.30 | 8.49 | 8.69 | 8.48 | 8.61 | 8.83 |

Table 11: The floating–point rate in Mflop/s achieved in *each* CM–200 processor for the multiplication of two square matrices.

performance of the matrix–vector multiplication routine is the excessively short inner loop (vector length 2) and the page faults in the inner loop with a column major layout. Because of the column major layout, there would be no page faults in the inner loop if the inner product routine had been used. The performance data reported in Table 2 would apply. The performance numbers in Table 2 are all based on a memory stride of 1 for each vector. A _DOT–like loop ordering for vector–matrix multiplication would yield a somewhat better performance than given in Table 2, as explained in Section 5. The CMSSL_DGEMV routine yields higher performance than CMSSL_DDOT routine for all cases and loop orderings we considered.

Table 11 and Figure 15 give performance data for the multiplication of square matrices in each processing node. The dips in the performance are due to the limitation on the vector length (maximum 22 rows) and the limitation on chaining of vector instructions (either 8 or none). The matrices in Table 11 and Figure 15 are sufficiently small that page faults do not have any (significant) impact. Table 12 and Figures 16 and 17 show the performance for the multiplication of a pair of matrices in which one is square, the other rectangular. The effect of choosing either the loop over the $P$–axis or the $R$–axis as the innermost loop is apparent. In the absence of page faults, the loop shall be chosen such that the length of the inner loop is maximized. When there are fewer columns in $B$ than rows in $A$, i.e., $R < P$, then the loop for the $R$–axis should be the outermost loop, and the loop for the $P$–axis should be the innermost loop. Thus, for a fixed matrix $A$, as the number of columns of $B$ increases, $R < P$, the performance is flat until $R = P$, then it increases with the number of columns of $B$. A similar behavior is apparent when $B$ is square, and the number of rows of $A$ is increased, as seen from Figure 17.

Choosing loop order for best performance must take the vector length, various overheads, and DRAM page faults into account. For the matrix multiplication $C \leftarrow A \times B$, the stride along the $P$-axis of $A$ is $A_0$ and the stride along the $Q$-axis of $A$ is $A_1$. The stride along the $Q$-axis of $B$ is $B_0$ and the stride along its $R$-axis is $B_1$. Finally, the stride along the $P$-axis of $C$ is $C_0$ and the stride along its $R$-axis is $C_1$. Since all our routines are designed for multiple instance computation, the strides of the $P$-axis of $A$ and $C$ need not be the same, neither need the strides along the $Q$-axis of $A$ and $B$, or the $R$-axis of $A$ and $C$, be the same. The stride expressed in memory addresses is also dependent upon the data type. This fact is accounted for in the strides $A_0$, $A_1$, $B_0$, $B_1$, $C_0$ and $C_1$. But, the number of 32–bit memory words loaded from and stored to memory must be accounted for explicitly which is done through a factor $S$, where $S = 1$ for 32–bit precision, $S = 2$ for 64–bit precision. The sum of the pipeline start–up and shut–down cost is denoted $\tau$. Table 13 summarizes the cycle estimates for matrix–vector multiplication, with the loop

| $P = Q$ | $R$ | SGEMM | DGEMM | $P$ | $Q = R$ | SEGMM | DGEMM |
|---|---|---|---|---|---|---|---|
| 4 | 4 | 2.33 | 1.89 | 4 | 4 | 2.33 | 1.89 |
| 16 | 4 | 12.10 | 6.88 | 16 | 4 | 5.88 | 3.79 |
| 64 | 4 | 16.80 | 8.69 | 64 | 4 | 7.59 | 4.59 |
| 256 | 4 | 17.70 | — | 256 | 4 | 8.20 | 4.82 |
| 1024 | 4 | — | — | 1024 | 4 | 8.43 | 4.90 |
| 4 | 16 | 5.88 | 3.84 | 4 | 16 | 12.10 | 6.88 |
| 16 | 16 | 12.10 | 6.88 | 16 | 16 | 12.10 | 6.88 |
| 64 | 16 | 16.80 | 8.69 | 64 | 16 | 14.00 | 7.51 |
| 256 | 16 | 17.70 | — | 256 | 16 | 14.50 | 7.61 |
| 1024 | 16 | — | — | 1024 | 16 | 14.30 | 7.59 |
| 4 | 64 | 7.67 | 4.59 | 4 | 64 | 15.70 | 8.15 |
| 16 | 64 | 14.00 | 7.43 | 16 | 64 | 15.70 | 8.15 |
| 64 | 64 | 16.80 | 8.69 | 64 | 64 | 16.80 | 8.69 |
| 256 | 64 | 17.70 | — | 256 | 64 | 16.90 | 8.68 |
| 1024 | 64 | — | — | 1024 | 64 | — | — |
| 4 | 256 | 8.21 | 4.81 | 4 | 256 | 14.30 | — |
| 16 | 256 | 14.30 | 7.51 | 16 | 256 | 17.10 | — |
| 64 | 256 | 16.80 | 8.69 | 64 | 256 | 17.70 | — |
| 256 | 256 | — | — | 256 | 256 | — | — |
| 1024 | 256 | — | — | 1024 | 256 | — | — |
| 4 | 1024 | 8.42 | 4.89 | 4 | 1024 | — | — |
| 16 | 1024 | 14.40 | 7.55 | 16 | 1024 | — | — |
| 64 | 1024 | — | — | 64 | 1024 | — | — |
| 256 | 1024 | — | — | 256 | 1024 | — | — |
| 1024 | 1024 | — | — | 1024 | 1024 | — | — |

Table 12: The floating–point rate in Mflop/s achieved in *each* CM–200 processor for the multiplication of one square and one rectangular matrix.

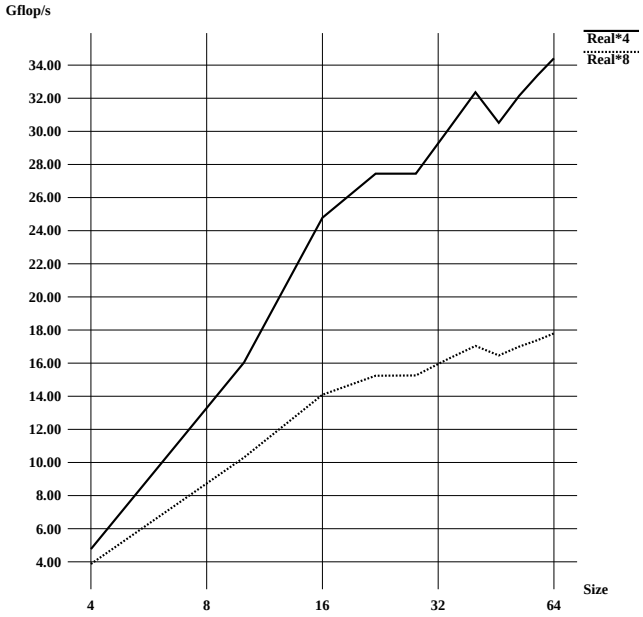


Figure 15: The aggregate performance for square matrix multiplication in 32–bit and 64–bit precision on a 2048 processor CM–200.

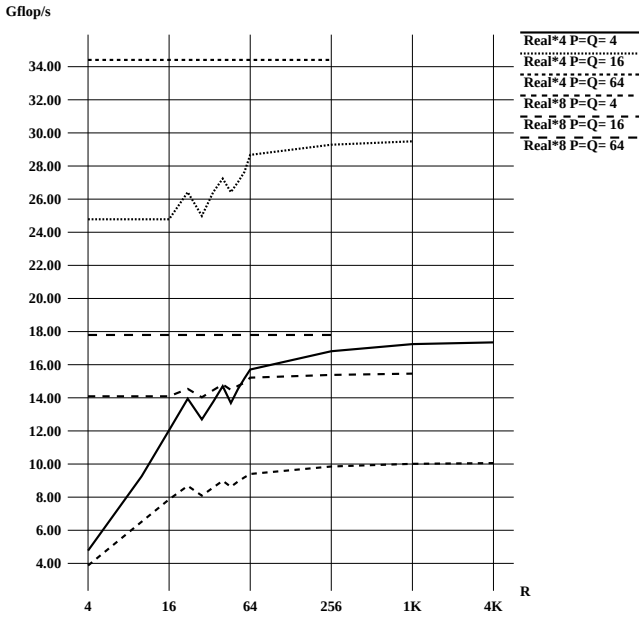


Figure 16: The aggregate performance for the multiplication of a $P \times P$ matrix by a $P \times R$ matrix in 32–bit and 64–bit precision on a 2048 processor CM–200.

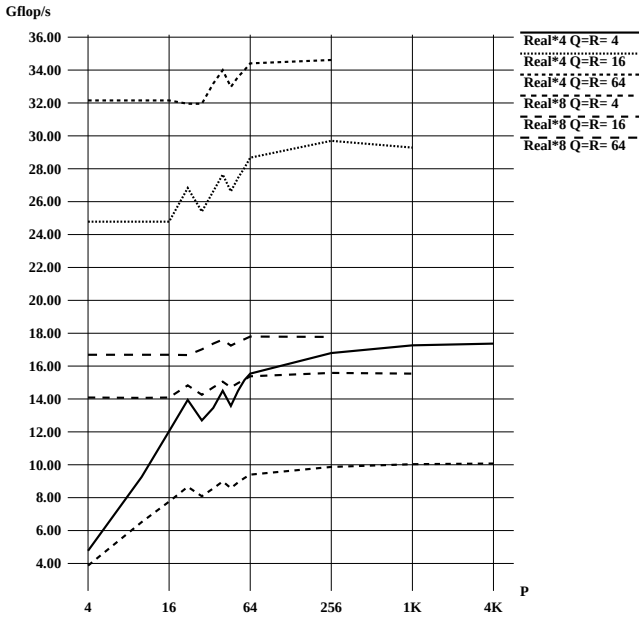


Figure 17: The aggregate performance for the multiplication of a $P \times Q$ matrix and a $Q \times Q$ matrix in 32–bit and 64–bit precision on a 2048 processor CM–200.

on the $P$–axis being the innermost loop. In addition to the entries in Table 13, there is also a one time cost $\sigma$ for the function call. The value of $\tau$ and $\sigma$ amounts to about 6 and 10 cycles, respectively.

For vector-matrix multiplication the roles of $A$ and $B$ are interchanged. More precisely, $A$ has taken the role of $B$ with the stride $A_1$ replacing that of $B_0$; $B$ has taken the role of $A$ with $B_1$ replacing $A_0$ and $B_0$ replacing $A_1$. $C$ is accessed by rows instead of columns, and $C_1$ replaces $C_0$.

The choice of inner and outermost loops for matrix–matrix multiplication is based on the above expressions, acknowledging that with the $P$–axis being the innermost, $R$ calls are

| Operand | DRAM page faults | Load/Store Cycles | Overhead |
|---|---|---|---|
| $A$ | $(\lfloor \frac{A_0 * 22}{\mathcal{P}} \rfloor \lfloor \frac{P}{22} \rfloor + \lfloor \frac{A_0 * P \bmod 22}{\mathcal{P}} \rfloor) Q + (\lceil \frac{Q}{8} \rceil + Q) \lceil \frac{P}{22} \rceil$ | $P * Q * S$ | $8 * \lfloor \frac{P}{22} \rfloor + \tau \lceil \frac{P}{22} \rceil +$ |
| $B$ | $((1 + \lfloor \frac{B_0 * 8}{\mathcal{P}} \rfloor) \lfloor \frac{Q}{8} \rfloor + Q \bmod 8) \lceil \frac{P}{22} \rceil$ | $Q * S * \lceil \frac{P}{22} \rceil$ | $+\{8 \lceil \frac{P}{22} \rceil + \tau \lceil \frac{P}{22} \rceil\}$ if $(Q \bmod 8 \neq 0$ $\& Q > 8)$ |
| $C$ | $\lfloor \frac{C_0 * 22}{\mathcal{P}} \rfloor \lfloor \frac{P}{22} \rfloor + \lfloor \frac{C_0 * P \bmod 22}{\mathcal{P}} \rfloor + \lceil \frac{P}{22} \rceil$ | $3 * P * S$ | $2\tau \lceil \frac{P}{22} \rceil$ |

Table 13: Estimation of the number of cycles required for matrix–vector multiplication.

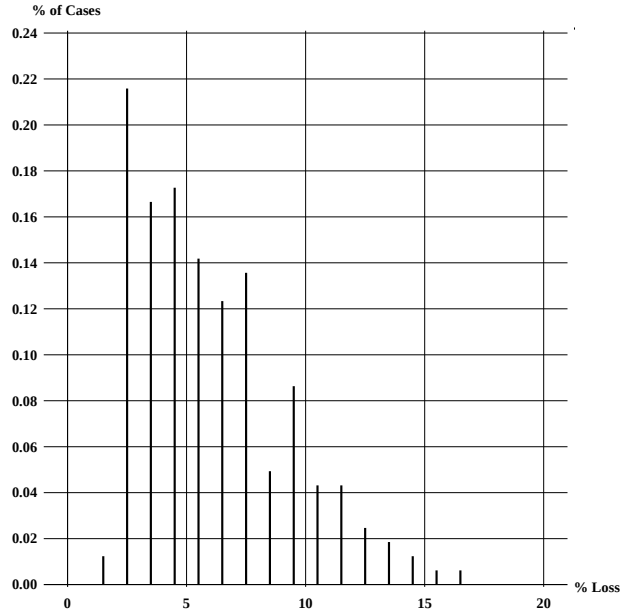


Figure 18: The percentage performance loss due to nonoptimal choice of loop ordering in matrix–matrix multiplication local to *each* CM–200 processor.

required, while for the $R$–axis being innermost, $P$ calls are required. The losses due to a nonoptimal choice of loop order are illustrated in Figure 18. The optimal choice is indeed made in more than 99.8% of the cases for which the sum of the length of the axes is at most 60. The performance loss due to an incorrect choice is at most $\sim 15\%$ and typically less than 1%.

## 6.1 Rank–$n$ updates

A rank–$n$ update is a generalization of a rank–1 update. Rank–$n$ updates are used in many block algorithms such as block LU or QR factorization, as well as in the solution of triangular systems of equations. In a rank–$n$ update the outer–product between $n$ column vectors and $n$ row vectors are computed at once. A rank–$n$ update is a matrix–matrix multiplication in which the inner dimension typically is significantly less than the outer dimensions, i.e., $Q \ll P, R$. Typical values of $n$ in block algorithms are in the range 2 – 30, with the larger values used mostly in very large problems [9]. The gain in performance by using a rank–$n$ update instead of a rank–1 update can be quite significant. A local rank–$n$ update may yield five times higher performance than a rank–1 update in the CM–200 LBLAS.

In the LBLAS there is no special interface provided for rank–$n$ updates. The CMSSL_GEMM interface is used. Tables 14 and 15 and Figure 19 give the performance for rank–$n$ updates on square matrices for $n$ in the range 1 – 256. The relative performance enhancement

| CMSSL_SGEMM | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of rows | Rank | | | | | | | | | | |
| & columns | 1 | 2 | 3 | 4 | 5 | 8 | 16 | 32 | 64 | 128 | 256 |
| 2 | 0.34 | 0.54 | 0.79 | 1.02 | 1.23 | 2.04 | 3.47 | 5.29 | 7.18 | 8.71 | 9.21 |
| 3 | 0.50 | 0.81 | 1.16 | 1.46 | 1.74 | 2.86 | 4.68 | 6.89 | 9.01 | 10.30 | 11.20 |
| 4 | 0.69 | 1.10 | 1.56 | 1.96 | 2.32 | 3.74 | 3.26 | 8.36 | 10.50 | 11.60 | 12.50 |
| 5 | 0.84 | 1.34 | 1.88 | 2.35 | 2.77 | 4.39 | 6.79 | 9.35 | 11.40 | 12.60 | 13.50 |
| 8 | 1.23 | 1.96 | 2.72 | 3.38 | 3.94 | 5.98 | 8.77 | 11.40 | 13.20 | 14.50 | 15.20 |
| 16 | 2.02 | 3.20 | 4.34 | 5.27 | 6.05 | 8.53 | 11.50 | 13.80 | 15.50 | 16.50 | 17.00 |
| 32 | 2.49 | 3.91 | 5.18 | 6.18 | 7.00 | 9.66 | 12.40 | 14.60 | 16.00 | 16.70 | 17.20 |
| 64 | 3.26 | 4.93 | 6.40 | 7.52 | 8.28 | 11.10 | 13.80 | 15.60 | 16.70 | 17.40 | 17.70 |
| 128 | 3.80 | 5.43 | 6.83 | 7.82 | 8.64 | 11.70 | 14.20 | 15.90 | 16.90 | 17.50 | 17.80 |
| 256 | 4.15 | 5.60 | 7.06 | 8.12 | 8.92 | 12.00 | 14.40 | 16.00 | 16.90 | 17.40 | 17.70 |
| 512 | 4.29 | 5.75 | 7.22 | 8.27 | 9.07 | 12.10 | 14.40 | 15.90 | 16.80 | 17.30 | — |

Table 14: The floating–point rate in Mflop/s achieved in *each* CM–200 processor for rank–$n$ updates, 32–bit precision.

achieved by a rank–$n$ update over a rank–1 update is shown in Figure 20. The relative enhancement is greater for small matrices than for large matrices. For instance, the performance enhancement of a rank–32 update over a rank–1 update, is a factor of about 5.9 for $32 \times 32$ matrices. For $256 \times 256$ matrices the improvement is a factor of 3.9 in 32–bit precision. The performance improvement in 64–bit precision for the same matrix sizes is 4.7 and 3.5, respectively.

# 7 Summary

We have presented a local BLAS (LBLAS) for distributed memory architectures and languages with an array syntax. The LBLAS are designed to perform the operations

$$C^{op_C} \leftarrow C^{op_C} \pm A^{op_A} \times B_{op_B}$$

where $A$, $B$, and $C$ are real or complex matrices in 32–bit or 64–bit precision and of any shape and memory layout that can be specified for arrays with an arbitrary number of axis. The superscript for $A$ and $C$ designates either the matrix or its transpose, while for $B$, the superscript designates one of four options: normal, transpose, complex conjugate, or Hermitian.

Given the architectural features of the CM–200 processor, the peak performance for BLAS routines is limited by memory accesses. Thus, the expected performance of the _NRM2 routines is twice that of the corresponding _DOT routine, which, in turn, is twice that of the _AXPY routine. The peak efficiency achieved for the DNRM2, DDOT and DAXPY

| CMSSL_DGEMM | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| No. of rows & columns | Rank | | | | | | | | | | |
| | 1 | 2 | 3 | 4 | 5 | 8 | 16 | 32 | 64 | 128 | 256 |
| 2 | 0.30 | 0.50 | 0.71 | 0.92 | 1.10 | 1.75 | 2.71 | 3.73 | 4.59 | 5.04 | 5.39 |
| 3 | 0.43 | 0.73 | 1.05 | 1.32 | 1.54 | 2.37 | 3.52 | 4.66 | 5.47 | 6.00 | 6.32 |
| 4 | 0.54 | 0.95 | 1.32 | 1.63 | 1.90 | 2.85 | 2.00 | 5.28 | 6.06 | 6.61 | 6.92 |
| 5 | 0.65 | 1.13 | 1.56 | 1.91 | 2.22 | 3.25 | 4.59 | 5.74 | 6.53 | 7.04 | 7.33 |
| 8 | 0.91 | 1.57 | 2.12 | 2.57 | 2.95 | 4.12 | 5.52 | 6.61 | 7.37 | 7.82 | 8.07 |
| 16 | 1.37 | 2.30 | 3.02 | 3.58 | 4.04 | 5.29 | 6.65 | 7.64 | 8.26 | 8.61 | 8.80 |
| 32 | 1.66 | 2.66 | 3.43 | 4.02 | 4.43 | 5.73 | 7.00 | 7.87 | 8.40 | 8.68 | 8.84 |
| 64 | 2.00 | 3.12 | 3.91 | 4.48 | 4.94 | 6.26 | 7.44 | 8.22 | 8.67 | 8.92 | 9.04 |
| 128 | 2.23 | 3.27 | 4.10 | 4.68 | 5.13 | 6.44 | 7.56 | 8.28 | 8.69 | 8.91 | 9.03 |
| 256 | 2.36 | 3.38 | 4.21 | 4.79 | 5.23 | 6.52 | 7.59 | 8.28 | 8.67 | 8.88 | 8.99 |

Table 15: The floating–point rate in Mflop/s achieved in *each* CM–200 processor for rank–$n$ updates, 64–bit precision.

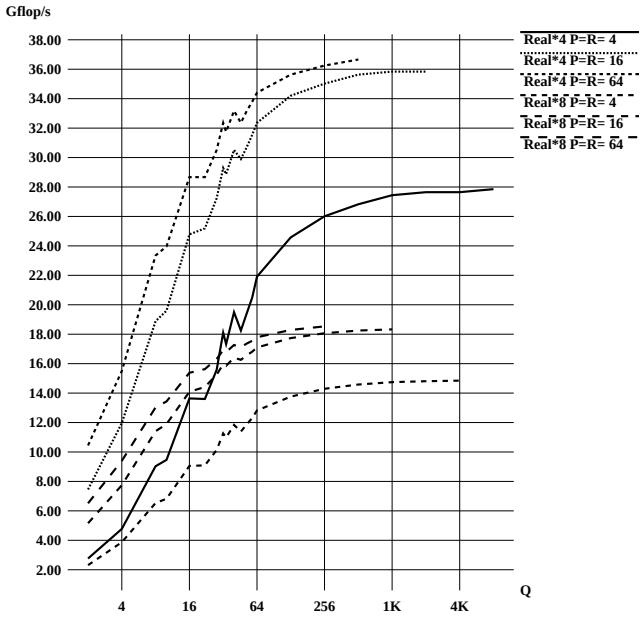


Figure 19: The aggregate performance for rank–n updates of $P \times P$ matrices in 32–bit and 64–bit precision on a 2048 processor CM–200.

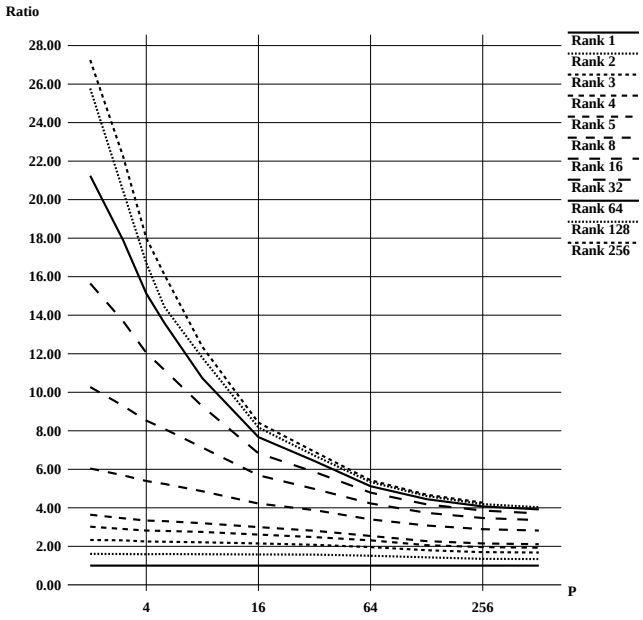


Figure 20: The performance ratio between rank–$n$ and rank–1 updates of $P \times P$ matrices local to a CM–200 processor, 32–bit precision.

routines is 80%, 85%, and 100%, respectively. These efficiency estimates are based on stores requiring two cycles, which is conservative. The peak measured performance is 8.04 Mflop/s, 4.25 Mflop/s, and 2.49 Mflop/s per processor. Clearly, for algorithms that can make use of either the DDOT routine or the DAXPY routine, the former should be chosen. The efficiency in 32–bit precision is slightly less. The vector length for half of peak performance is 35, 20 and 10 for DNRM2, DDOT and DAXPY, respectively.

Matrix–vector and vector–matrix multiplication can fully exploit the arithmetic capability of the processor architecture in 32–bit precision. In 64–bit precision the fact that the path to memory is 32–bits wide limits the peak performance to half of the peak arithmetic capability of the CM–200. The peak measured floating–point rate is 18.0 Mflop/s per processor in 32–bit precision (90% efficiency), and 9.2 Mflop/s per processor in 64–bit precision (92% efficiency).

Our rank–1 update routines are based on _AXPY routines and, at best, achieve a quarter of the peak processor performance. The measured peak floating–point rate is 4.2 Mflop/s per processor in 32–bit precision and 2.3 Mflop/s per processor in 64–bit precision. The choice between row-wise or column-wise _AXPY operations is made in the same way as for matrix–matrix multiplication.

Our LBLAS matrix–matrix multiplication routines are based on the level–1 and level–2 LBLAS. A level–1 LBLAS routine is only used when the matrix shapes are such that a single call to such a routine suffices to carry out the requested operation. When the matrix–matrix multiplication consists of three nested loops with more than one iteration

in each loop, the level–2 LBLAS are used and the loop ordering determined at run–time based on an estimate of page faults, pipeline losses, and looping overhead. The optimum choice is indeed made in more than 99.8% of the cases for which the sum of the length of the axes is, at most, 60. For all possible axes' lengths, the optimum choice is made in an even higher fraction of the cases. The performance loss due to an incorrect choice is at most $\sim 15\%$ and typically less than 1%.

We also show that the performance gain in using a rank–$n$ update instead of a rank–1 update may be close to a factor of 6, with a factor of 3 - 4 being more typical.

**Acknowledgments**

Many people have contributed to the LBLAS on the Connection Machine. We would like to thank in particular Mark Bromley, Tim Harris, Robert L. Krawitz, Woody Lichtenstein, Bob Lordi, Douglas MacDonald, Kapil K. Mathur and Bill Nesheim. The authors also wish to thank the referees for many valuable suggestions that helped improve the presentation, and for their timely review.

# References

[1] Jack J. Dongarra, Jeremy Du Croz, Iain Duff, and Sven Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. Technical Report Reprint No. 1, Argonne National Laboratories, Mathematics and Computer Science Division, August 1988.

[2] Jack J. Dongarra, Jeremy Du Croz, Sven Hammarling, and Richard J. Hanson. An Extended Set of Fortran Basic Linear Algebra Subprograms. Technical Report Technical Memorandum 41, Argonne National Laboratories, Mathematics and Computer Science Division, November 1986.

[3] Jack J. Dongarra, Peter Mayes, and Giuseppe Radicati di Brozolo. The IBM RISC system/6000 and linear algebra operations. LAPACK working note 28. Technical Report CS-90-122, University of Tennessee, Department of Computer Science, December 1990.

[4] S. Lennart Johnsson. *Compilation Techniques for Novel Architectures*, chapter *Language and Compiler Issues in Scalable High Performance Libraries*. Springer Verlag, 1993. Harvard University Technical Report TR-18-92.

[5] S. Lennart Johnsson and Kapil K. Mathur. Distributed BLAS. Technical report, Thinking Machines Corp., 1992. In preparation.

[6] S. Lennart Johnsson and Kapil K. Mathur. Distributed level 1 and level 2 BLAS. Technical report, Thinking Machines Corp., 1992. In preparation.

[7] Bo Kågström, P. Ling, and Charles 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. North Holland, 1991.

[8] C.L. Lawson, R.J. Hanson, D.R. Kincaid, and F.T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM TOMS*, 5(3):308–323, September 1979.

[9] Woody Lichtenstein and S. Lennart Johnsson. Block cyclic dense linear algebra. *SIAM Journal of Scientific Computing*, 14(6):1257–1286, 1993.

[10] P. Ling. A set of high performance level–3 BLAS structured and tuned for the IBM 3090 VF and implemented in Fortran 77. Technical Report UMINF-179.90, Department of Information Processing, University of Umeå, 1992.

[11] Kapil K. Mathur and S. Lennart Johnsson. Multiplication of matrices of arbitrary shape on a Data Parallel Computer. *Parallel Computing*, 20(7):919–951, July 1994.

[12] Michael Metcalf and John Reid. *Fortran 90 Explained.* Oxford Scientific Publications, 1991.

[13] Pelle Olsson and S. Lennart Johnsson. A dataparallel implementation of explicit methods for the three-dimensional compressible Navier-Stokes equations. *Parallel Computing*, 14(1):1–30, 1990.

[14] Thinking Machines Corp. *CM–200 Technical Summary*, 1991.

[15] Thinking Machines Corp. *CM–5 Technical Summary*, 1991.

[16] Thinking Machines Corp. *CM Fortran optimization notes: slicewise model, version 1.0*, 1991.

[17] Thinking Machines Corp. *CM Fortran Reference Manual, Version 2.1*, 1993.

[18] Thinking Machines Corp. *CMSSL for CM Fortran, Version 3.1*, 1993.