LOCAL BASIC LINEAR ALGEBRA SUBROUTINES (LBLAS) FOR THE CM–5/5E

David Kramer
193 ROCK HARBOR LANE
FOSTER CITY, CA 94404

S. Lennart Johnsson
AIKEN COMPUTATION LAB
HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS 02138
AND
UNIVERSITY OF HOUSTON
HOUSTON, TEXAS 77204-3475

Yu Hu
AIKEN COMPUTATION LAB
HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS 02138

Summary

The Connection Machine Scientific Software Library (CMSSL) is a library of scientific routines designed for distributed memory architectures. The BLAS of the CMSSL have been implemented as a two–level structure to exploit optimizations local to nodes and across nodes. This paper presents the implementation considerations and performance of the Local BLAS, or BLAS local to each node of the system. A wide variety of loop structures and unrollings have been implemented in order to achieve a uniform and high performance, irrespective of the data layout in node memory. The CMSSL is the only existing high–performance library capable of supporting both the data parallel and message passing modes of programming a distributed memory computer. The implications of implementing BLAS on distributed memory computers are considered in this light.

The International Journal of Supercomputer Applications and High Performance Computing,
Volume 10, No. 4, Spring 1996, pp. 300–335
©1996 Sage Publications, Inc.

Introduction

The Connection Machine systems provide high–performance computation on large, data–intensive problems and have found successful application in a diverse range of floating–point intensive applications. The Connection Machine Scientific Software Library (CMSSL) (Thinking Machines Corporation, 1993) is a software library designed for efficient use of distributed memory architectures, in particular the Connection Machine systems CM–2/200 and CM–5/5E. The CMSSL includes many algorithms commonly used in scientific computation, such as routines for dense and sparse matrix manipulation, linear systems solvers, eigensystem analysis, Fast Fourier Transforms and Differential Equation solvers. The fundament of many of these algorithms is a set of relatively simple linear algebra operations known as the Basic Linear Algebra Subroutines (BLAS) (Lawson et al., 1979a, 1979b). In order to achieve high overall performance, it is necessary to optimize these fundamental operations.

Highly optimized BLAS implementations are available on most high–performance computers. Initially, these routines comprised of operations on matrices of rank–1 (vectors). These routines, termed Level–1 BLAS, found widespread acceptance amongst developers and were incorporated into a wide array of software applications. The rise of pipelined computer architectures, particularly vector computers with a limited bandwidth to memory (one operand per cycle), revealed a need for codes which could retain a vector of data in the register set. The Level–2 BLAS were introduced (Dongarra et al., 1988a, 1988b) to operate on matrices of rank–2. More recently, the widespread introduction of architectures with hierarchical memory structures, particularly data caches, led to the design of the Level–3 BLAS (Dongarra et al., 1990a, 1990b). These codes operate on pairs of matrices, and are able to exploit a blocked memory access scheme. Blocked memory access schemes are becoming increasingly important also in noncached memory accesses due to the organization of memory chips (Comerford and Watson, 1992, Farmwald and Morning, 1992).

Distributed memory architectures introduce an added level of complexity into numerical operations such as the BLAS. Matrices may be distributed across individual processor memories in many ways. Furthermore, the matrix operands local to each processor may be laid out in a wide variety of ways. In order to achieve high performance at both the local and global levels, the CMSSL adopts a hierarchical BLAS strategy. At the global level, for any given operation, one of a variety of algorithms is selected (Mathur and Johnsson, 1994). For example, matrix–matrix multiplication may be performed using one of several systolic algorithms. Other algorithms, e.g., based on a broadcast of one or more operands can also be chosen. The selection is based on the shapes and sizes of the operands, their distribution across the memory units, often referred to as *layout*, the overall availability of memory on the machine, and estimates of the performance for the various alternatives. The global software layer for the BLAS on the CM–5/5E is referred to as the Distributed BLAS (DBLAS) (Johnsson and Mathur, 1992). The *ScaLAPACK* library (Choi et al., 1992) was introduced as a library for performing DBLAS computations based on specific data layouts and generic level–3 BLAS routines.

The DBLAS controls how data is moved between processing nodes and which operations are to be performed on each node at each step of operation. Irrespective of which algorithm is selected by the DBLAS, it is still necessary to perform operations on the segments of matrices local to each node. In order to optimize these operations, a second level of BLAS, the Local BLAS (LBLAS) was designed and implemented. The LBLAS for the Connection Machine System CM–2/200 were described by Johnsson and Ortiz in (Johnsson and Ortiz, 1992). This paper describes the implementation and performance of the LBLAS on the CM–5/5E.

The CMSSL is designed to support languages with array syntax. Various standards for such languages have been proposed, such as Fortran 90 (Metcalf and Reid, 1991) and High Performance Fortran (High Performance Fortran Forum, 1993). Currently the CMSSL supports two such languages, namely *CM Fortran* (CMF) which heavily influenced the design of HPF, and *C**, which is being used as a strawman language for the design of a standard for a parallel C language. The languages determine the data structures and their representations and the distribution of the operands upon which library routines are operating. The implementation of the languages determines what information is available to library routines, and heavily influences the design of interfaces to the library. Library interfaces not only must be consistent with the definition of the language, but should also be consistent with the spirit of the language (libraries are in effect extensions to a language), and provide sufficient information for a high degree of optimization. High resource utilization is one important justification for software libraries. In the CMSSL, array operands are passed in–place, and all parallel computation occurs in a data parallel or “SPMD” fashion.

The LBLAS has found application far beyond the scope of the DBLAS. Many applications have been written which exploit the locally optimized LBLAS. LBLAS calls within CMSSL are used in a number of algorithms, such as linear system solvers and banded system solvers. Though the LBLAS is accessible for “embarassingly” parallel computations through DBLAS interfaces, such calls incur an unnecessary overhead when it is known to the calling program that the distribution of the operands is such that the computations in fact are “embarassingly” parallel. The two tier approach to the BLAS on distributed memory architectures is justified both from an implementation point of view and from a user point of view. The latter is supported, at the moment, through public DBLAS interfaces and LBLAS interfaces internal to the CMSSL.

Section 1 of this paper outlines the type of operations that have been implemented in the LBLAS, and how these differ from traditional BLAS operations. In order to explain the performance and optimizations of the LBLAS, Section 2 briefly outlines the architecture of the CM–5/5E, placing particular emphasis on the features of the CM–5/5E nodes that affect LBLAS performance. Section 3 describes the architecture of the LBLAS. Implementation issues and measured performance of the Level–1 LBLAS are described in Section 4, while the Level–2 LBLAS are presented in Section 5. A brief discussion of Level–3 LBLAS is given in Section 6. Many of the decisions of loop partitioning and loop ordering for high performance can not be made until run–time. The mechanism for run–time selection of LBLAS kernels is discussed in Section 7. The paper concludes with a summary and discussion of results.

1 The scope of the LBLAS

1.1 Functionality

The functionality currently implemented in the LBLAS is summarized in Table 1.

The functionality of the LBLAS differs from the cor-

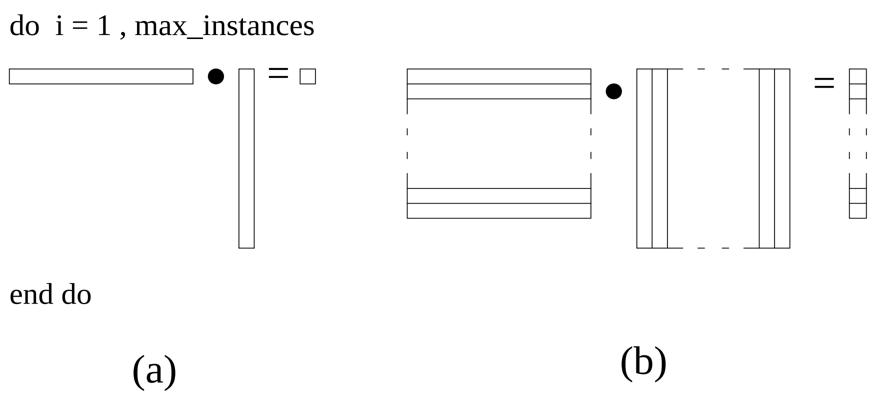


Figure 1: Implementation of a multiple–instance inner–product call using (a) looping over instance–axis and (b) one multiple–instance call.

Table 1: Functionality of the LBLAS on the CM–5/5E.

Function	Operations
_SCAL	Scalar–vector multiply
_AXPY	Vector plus scalar–vector multiply
_DOT	Inner product of two vectors
_SIP	Self inner product of one vector
_NRM2	Squared L_2 norm of a vector
_NRM2SQ	L_2 norm of a vector
_GER	Outer product of two vectors
_GEMV	Matrix–vector multiply
_GEVM	Vector–matrix multiply
_GEMM	Matrix–matrix multiply

responding standard BLAS in a few ways. One general difference is the lack of scaling variables. The reason for this difference is that the DBLAS, which the LBLAS supports, was designed without scaling variables for efficiency. In the DBLAS, scaling variables take the form of arrays with rank of one less than arrays representing vectors in the standard BLAS. On the Connection Machine systems, the Run–Time System distributes arrays evenly over the nodes, with the default subgrid shape having axes lengths of approximately equal extents. With this data distribution scheme, a large communication overhead may be incurred when scaling parameters are introduced in the DBLAS, since the array of scaling parameters is, in general, not aligned with any other operand. Explicitly aligning the array of scaling parameters with another operand array through the use of compiler directives may increase the storage requirements substantially. Therefore, scaling is left to the user in the current DBLAS and LBLAS implementations.

1.2 Multiple–Instance Operations

One distinguishing characteristic of the CMSSL is that it supports a *multiple–instance* interface. The need to perform the same operation on multiple independent operands arise in a wide variety of scientific problems, and in divide–and–conquer algorithms. An example is the solution of Navier–Stokes equations in fluid dynamics, which requires a matrix–vector multiplication at each grid point (Olsson and Johnsson, 1990). In the traditional implementation a number of loops would serially invoke a call to a matrix–vector product BLAS. In the multiple–instance formulation, the operations at each grid point are treated as a single call to larger problem. Figure 1 illustrates an inner–product operation on a problem which has an array with a single instance–axis of extent *max_instances*. In Figure 1 (a) a number of calls to the inner–product routine are required. In Figure 1 (b) a single function call is required. The benefit of this formulation is that within the library routine an additional degree of freedom is available for optimization. Moreover, this degree of freedom represents embarassingly parallel computations. We refer to the axis (axes) defining the elements of a single operand as the *problem–axis (axes)* for that operand, and the other axes as *instance–axes*. The problem–axis (axes) are always present, while instance–axes may be absent for a single–

instance computation. It is entirely possible, in fact very likely because of the data distribution rules used by CMF and the Connection Machine Run–Time System (Thinking Machines Corporation, 1994), that the axis with stride one in an array is an instance–axis. As it is often desirable to vectorize operations along the axis with stride one, the LBLAS are highly optimized for *vectorization along any available axis.* Also, the multiple–instance feature generally lowers the overhead associated with looping and temporary storage. Most routines in the CMSSL use the same interfaces for single–instance and multiple–instance computations.

Since any operand for the BLAS has either one or two problem–axes, an operand may have one, two, or three contiguous sets of instance–axes. In the DBLAS, these three sets are identified, and within each set the instance–axes are folded into a single axis, whenever there is more than one axes in the set. The problem–axes together with the folded instance–axis that is predicted to offer the best opportunities for performance optimization are passed to the LBLAS. The LBLAS are designed to handle the problem–axis (axes) and one instance–axis. The ordering and partitioning of the other instance–axes, if any, is made in the DBLAS calling the LBLAS. These tasks remain with the user if the LBLAS is called directly.

1.3 Definitions and notation

The functionality of the currently implemented LBLAS can be cast as a generalized matrix multiplication:

$$C^{op_C} \leftarrow \alpha C^{op_c} \pm A^{op_A} B^{op_B}, \quad \alpha \in 0, 1$$

where C, A and B are arrays with up to two problem–axes and one instance–axis each. 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 H is Hermitian (complex conjugate transpose) and C is complex conjugate. The combination of these that is implemented depends on which out of six *mode* parameters that is specified. The mode parameter defines how the operands are combined. The six modes are presented in Table 2, (where $\overline{B}$ implies the complex conjugate of B).

The LBLAS directly implement Level–1 and Level–2 functions, while Level–3 functions are implemented in terms of Level–1 and Level–2 LBLAS (as discussed in Section 6). The Level–3 _GEMM operation has been

Table 2: LBLAS modes.

Mode	Description	Operation
-2	Conjugate Sub	$C = C - A\overline{B}$
-1	Subfrom	$C = C - AB$
0	Noadd	$C = AB$
1	Addto	$C = C + AB$
2	Conjugate Add	$C = C + A\overline{B}$
3	Conjugate Noadd	$C = A\overline{B}$

implemented using the $M3$ method (Hingham, 1992) to achieve maximum performance on complex data.

The LBLAS support the four common floating–point data types, namely *single* (32–bit real), *double* (64–bit real), *single complex* (64–bit complex) and *double complex* (128–bit complex). The LBLAS contain specific optimizations for 32–bit and 64–bit operations. However, there is currently no specific optimization for complex data types. Complex LBLAS calls are handled as a sequence of calls to kernels which operate on real data.

The shapes of the local arrays for any LBLAS operation can be described in terms of four parameters, p, q, r and m (see Figure 2). p is the number of rows in each instance in A and C. q is the number of columns in each instance in A and rows in each instance in B. r is the number of columns in B and C. m is the number of instances in each of the arrays. The arrays defined by the instance–axes must be conforming, and the instances ordered in the same way for all operands.

Table 3 shows the problem–axes ranks of the operands C, A and B. Each of these operands has an instance–axis in addition to the ranks shown in the table. The permissible values of p, q and r are also shown. π, ξ and ρ indicate arbitrary values. Finally, Table 3 also shows the number of floating–point operations per memory reference for the supported LBLAS functions.

2 Architecture of the CM–5/5E

To understand the design and optimizations for the LBLAS described in this paper, some familiarity with the CM–5/5E architecture (Thinking Machines Corporation, 1991) is required. The CM–5/5E system was designed for up to 16,384 processing nodes, each with its own memory (see Figure 3). A collection of nodes,

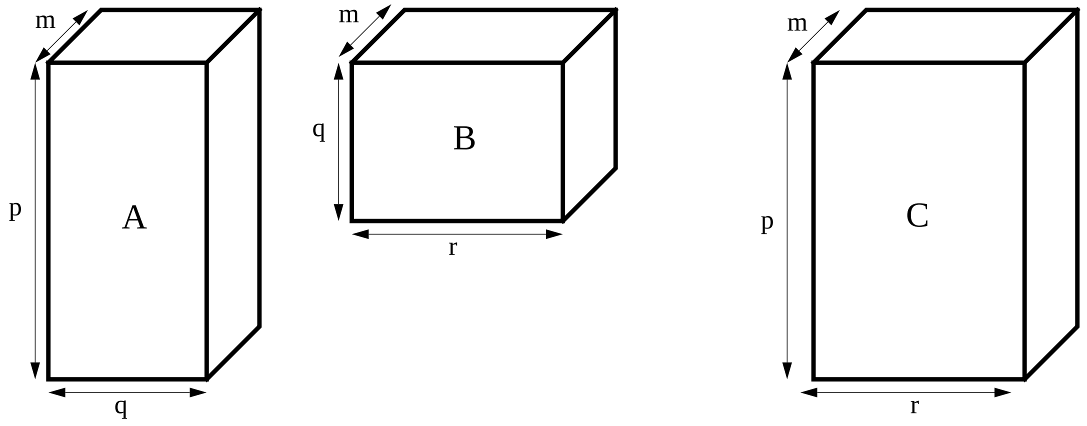


Figure 2: The shapes of the operands C, A and B and the meaning of the parameters p, q, r and m.

Table 3: LBLAS operations and matrix ranks (including instance–axis) of the operation $C^{op_C} \leftarrow \alpha C^{op_c} \pm A^{op_A} B^{op_B}, \quad \alpha \in 0, 1$.

Operation	Description	Rank of C	Rank of A	Rank of B	p	q	r	Flops/ mem ref
Level–1								
_SCAL	Scale	2	1	2	1	1	ρ	$\frac{1}{2}$
_AXPY	Scale and add	2	1	2	1	1	ρ	$\frac{2}{3}$
_DOT	Inner–product	1	2	2	1	ξ	1	1
_SIP	Self inner–product	1	2	2	1	ξ	1	2
_NRM2	L_2 Norm	1	2	-	1	ξ	1	2
_NRM2SQ	L_2 Norm with $sqrt$	1	2	-	1	ξ	1	2
Level–2								
_GER	Outer–product	3	2	2	π	1	ρ	1
_GEMV	Matrix–vector multiply	2	3	2	π	ξ	1	2
_GEVM	Vector–Matrix multiply	2	2	3	1	ξ	ρ	2
_GEMM	Matrix–matrix multiply	3	3	3	π	ξ	1	2

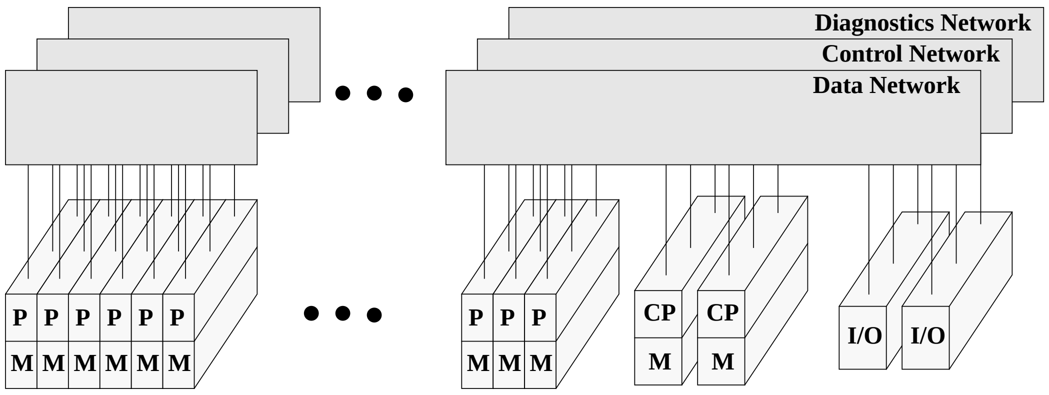


Figure 3: Overview of the CM–5/5E system.

known as a *partition*, is supervised by a Control Processor (CP), although the nodes may operate independently in MIMD mode. Each node is connected to two low latency, high bandwidth interconnection networks; the *Data Network* and the *Control Network*. The Data Network is generally used for point–to–point interprocessor communication, and the Control Network for operations such as synchronization and broadcasting. A third network, the *Diagnostics Network*, which is not available to the user, is used for ensuring the status of the system. As the LBLAS performance is dependent only on nodal performance and not on that of the networks, these will not be further discussed in this paper.

Figure 4 illustrates the architecture of a single node of a CM–5/5E. The node can be configured with four Vector Units (VUs) or no vector units. CMSSL exists for both configurations, but only the version for the CM–5/5E with VUs is highly optimized. The non–VU version of the LBLAS is not discussed in this paper. The four VU's are memory mapped into the node processor address space. It serves as a controller and coordinator for the VUs. Each VU consists of interfacing and control units as well as a Register File. The Register File can be viewed as sixty four 64–bit words, or as one hundred and twenty eight 32–bit words. The Register File in each VU is reconfigurable into sets of vector registers. For example, in 64–bits the Register File can be viewed as a single vector register of depth 64, two vector registers of depth 32 each, four register of depth 16 each, etc. The VU instruction set supports a four–stage pipeline. The maximum vector length is 16.

One significant restriction in the VU instruction set is the lack of support for indirect addressing of registers. This results in a need to explicitly code loop bounds into the kernels. Kernels with similar structures but which differ in loop extents or register allocations need to be individually coded and optimized. This results in a large number of distinct kernels.

The clock frequency of the node processor of the CM–5E is 40 MHz, and the VUs operate at half this frequency, which matches the clock rate of the memory. Each VU contains an ALU that can perform a multiply–add or a multiply–subtract operation on 64–bit floating–point or integer operands per clock cycle. Thus, the peak performance of each VU is 40 Mflops/s and the performance of each node is 160 Mflops/s. The VUs also support operations on 32–bit operands, but at the same performance level as for 64–bit operands.

Each VU is designed to address up to 128 Mbytes. The memory size per node is tied to the density of the memory chips used. With 4 Mbit memory chips each VU has 8 Mbytes of memory, while the use of 16 Mbit memory technology yields a VU memory of 32 Mbytes. Each VU has a single 64–bit wide data path to memory, giving peak memory bandwidth per node of 640 Mbytes/sec.

In order to maintain a small package, memory chips are organized as a matrix with row and column addresses being multiplexed onto the same pins. Row accesses are buffered allowing for single cycle accesses to columns, while row accesses require several cycles. The VU memory is dynamic RAM, with a DRAM page size of either

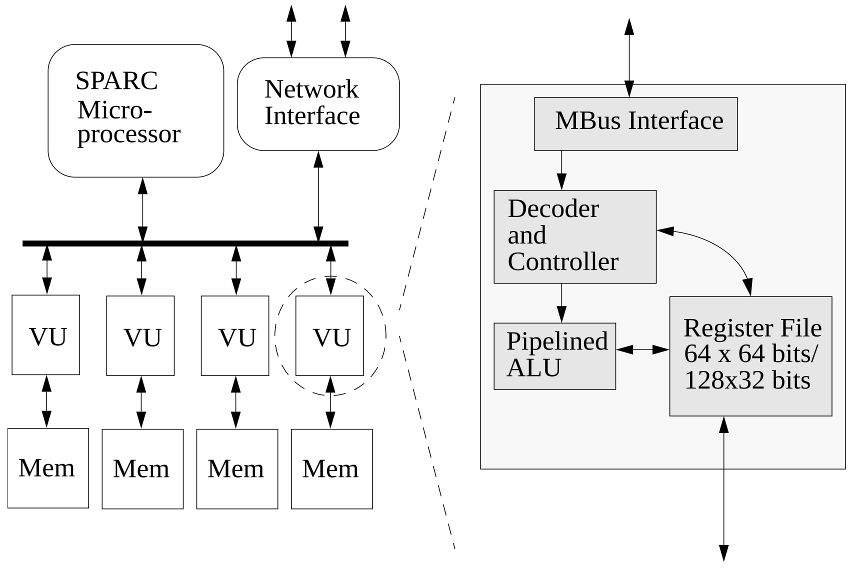


Figure 4: Overview of the CM–5/5E Node with Vector Units.

8 or 16 Kbytes. If the VU accesses two successive memory locations which are not on the same DRAM page, a DRAM *page fault* occurs. If a DRAM page fault occurs, the pipeline is stalled for 5 VU cycles, and hence it is desirable to implement looping structures within the LBLAS which minimize DRAM page faults.

Another performance hazard arising from the node architecture occurs because the VU and its associated memory is mapped into the memory space of the node processor. The node processor has a direct mapped Translation Lookahead Buffer, TLB, which generates the physical memory locations of the data. With current generations of the CMOST (Connection Machine Operating System), each of 64 TLB pages addresses 4 Kbytes of data. If the data being addressed is on a page not in the TLB then the pipeline is stalled for approximately 25 VU cycles. To compound this problem, *TLB thrashing* can occur. Consider a code which accesses l successive rows of data, in a loop. If the memory stride between rows is greater than 4 Kbytes, then a TLB miss occurs with each row access on the first iteration through the loop. If l is greater than the number of available TLB pages then the TLB is not able to hold all of the required pages concurrently. In this case, a TLB miss may occur as often as every vector instruction, or even every memory access. This severely degrades performance, and a looping structure that minimizes the risk for TLB thrashing is highly desirable

Operations on 32–bit data requires special attention for optimum performance on the CM–5/5E. The memory units are 64–bit wide. 32–bit data is stored in half words. But, each load always brings in a 64–bit word, and every store is a 64–bit store. 32–bit stores are carried out as a read–modify–write operation. A read–modify–write instruction requires 3.5 cycles. Thus, if two 32–bit results can be stored in the same 64–bit word, then the store can be made in one cycle, or half a cycle per 32–bit word, a speedup by a factor of seven compared to storing the 32–bit words individually. Using both 32–bit operands in a 64–bit word loaded into the Register File reduces the effective rate of a 32–bit load to half a cycle.

3 The LBLAS architecture

3.1 Execution control

The control of the execution of the LBLAS is divided among the Control Processor, the node processor, and the Vector Units. The LBLAS support two data parallel languages; CMF, and C*. The LBLAS also support three programming models: CMF or C* with a global address space, CMF or C* with a nodal address space using message passing for internode communication, and CMF or C* in global/local mode. The global/local mode corresponds to the extrinsic procedures option in High Performance Fortran (High Performance Fortran Forum, 1993).

The LBLAS code resides in the memory system of the CP. Array data resides in the memory of the VUs, while scalar data resides in the memory of the CP. This rule implies that, for instance, the result of a single inner–product resides in CP memory while the input arrays resides in VU memory. But, the result of a multiple–instance inner–product is an array, which resides in VU memory. Note however, that the result array is of rank one less than the input operands and thus, in general, will have a distribution different from that of the input operands.

Upon a call to a routine in the LBLAS, the CP invokes the execution of the code blocks on the node processors it manages. All node processors execute the same code block in a "SPMD" fashion. Synchronization of node processors takes place upon completion of the execution of each code block. The node processors simultaneously perform a number of tasks, such as the determination of array addresses, strides, and loop bounds. The node processors also determine which particular set of kernels shall be executed, and the order of execution. As the 'best' kernel to use in any situation depends on the shape of the operands and their allocation in memory, which can only be determined at run–time, this decision of which collection of kernels to use can only be made at run–time. Selection of a suite of non–optimal kernels for given operands can result in a performance degradation by more than an order of magnitude; factors of three to five are by no means rare. The run–time selection mechanism must be accurate and fast, so as not to constitute an undue overhead on the total running time. The decision is made on the basis of the lengths and strides of the problem– and instance–axes (see Section 7).

The node processor receives information from the CP that includes addresses, strides, and loop bounds. Much of this information is cached at the node processor, which then performs kernel selection and loop control. The VUs execute the arithmetic instructions contained in the loop body, under control of the node processors. Data is loaded from VU memory into VU registers, operated upon, and then results are stored back to VU memory.

3.2 Scheduling

The scheduling of operations take three levels of memory hierarchy into account: registers, DRAM pages and the TLB. In addition, for 32–bit operands the computations are organized to take advantage of both operands in a 64–bit word during load operations whenever possible, and in particular, schedules are sought that produce in succession both operands for a 64–bit word store. Each level of the memory hierarchy corresponds to a loop–nest. Thus, three loop–nests are needed to cover the three levels of the memory hierarchy. The number of iterations in a given loop corresponds to a partitioning of the corresponding loop in the next outer loop–nest. The loop–nests correspond to partitioning of the index space, as illustrated in Figure 5. The extents of the axes in loops in the innermost loop–nest define subdomains of the index space often called *register tiles* or simply tiles. In the LBLAS, a tile is two– or three–dimensional. The optimization consists of:

- determining the size and shape of a tile,
- determining the order in which tile axes are treated within a tile,
- determining the order in which tiles are treated.

As a rule, maximizing the size of a tile is a good idea. However, there are exceptions when more than one tile is required to cover the index space. In general, tiles of several sizes and shapes are required to cover the index domain. Tiles are constrained to have a size at most equal to the size of the Register File. Moreover, in order to reduce looping overhead, using tile axes extents that are powers–of–two is beneficial.

All LBLAS codes have a choice of axis for the innermost loop; the axis of vectorization. The LBLAS interface supports two potential problem–axes, and an instance–axis for each operand. Kernels have been written which are optimized to vectorize over any of these axes. In the case of the Level–1 BLAS, the choice is between vectorization over the single problem–axis, or the instance–axis. The Level–2 BLAS have several choices as to the ordering of the loops over the problem–axes, as well as the instance–axis. The ideal loop order for the

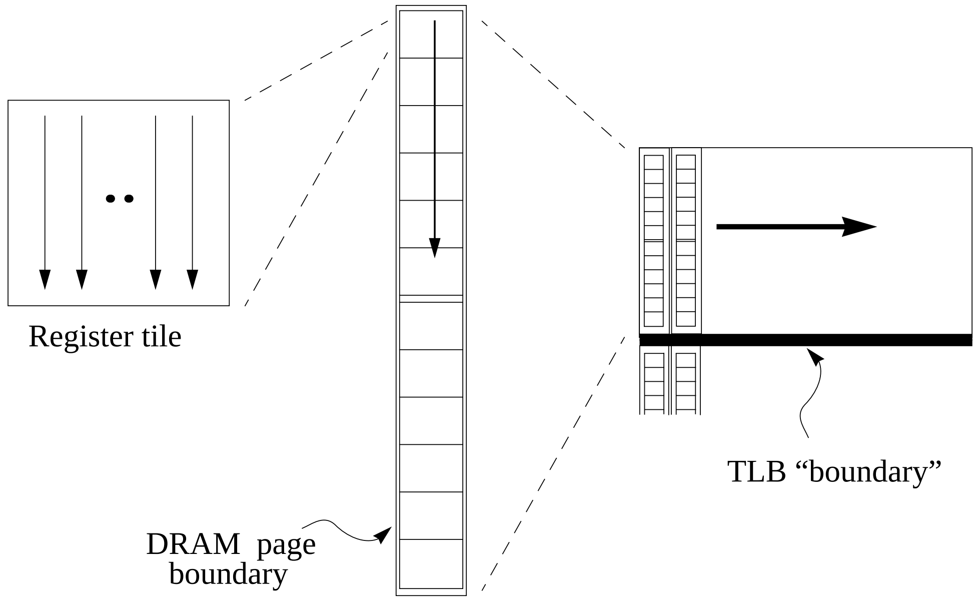


Figure 5: Tiling of the index space for optimum locality of reference.

tile with respect to performance can be derived from its shape and knowledge of the loop overheads and pipeline losses. Much of the loop overhead is determined by the node processor's ability to implement the required loop control and loop index updates concurrently with the VUs handling of the loop body. Thus, it is, for instance, better to partition a loop of length nine into a loop of length five and one of length four, than one of length eight and one of length one. The optimal order in which tiles are treated is determined based on load and store operations associated with each tile, DRAM page faults, and TLB misses.

In the design and implementation of the LBLAS, in addition to striving for high peak performance, considerable attention was devoted to achieving low overheads, i.e., minimizing the problem size required for half of peak performance, $n_{\frac{1}{2}}$ (Hockney and Jesshope, 1981).

In addition to loop partitioning and selecting loop orders for high performance, loop unrolling is also used to enhance performance. The following sections discuss functionality, design, implementation and performance of individual LBLAS functions.

4 Level–1 LBLAS

4.1 Functionality

Six Level–1 LBLAS functions have been implemented on the CM–5/5E. These are _SCAL, _AXPY, _DOT, _SIP, _NRM2, and _NRM2SQ. The first three functions differ from the corresponding standard BLAS by the lack of a scaling variable and the inclusion of a multiple–instance capability. The same differences exist for _NRM2SQ compared to the standard BLAS L_2–norm. _NRM2 is identical to _NRM2SQ, except it does not include a square root computation. _SIP computes a self inner–product. The functionality of _NRM2 and _SIP is required to support standard BLAS functionality in the DBLAS. Because the LBLAS support six modes, the definition of the _NRM2, namely $C \leftarrow C + AA^H \{C \in \Re\}$ has been extended to $\Re\{C\} \leftarrow \Re\{C\} + AA^{op_A}$.

_SIP performs the inner–product of a vector with itself, i.e., $C \leftarrow C + AA^{op_A}\{C \in \mathcal{C}\}$. This function differs from _NRM2 for complex data in that the _NRM2 function computes only the real value of the result, while _SIP computes both the real and imaginary components of the result. We have found this to be a useful optimization when a self inner–product of complex data is

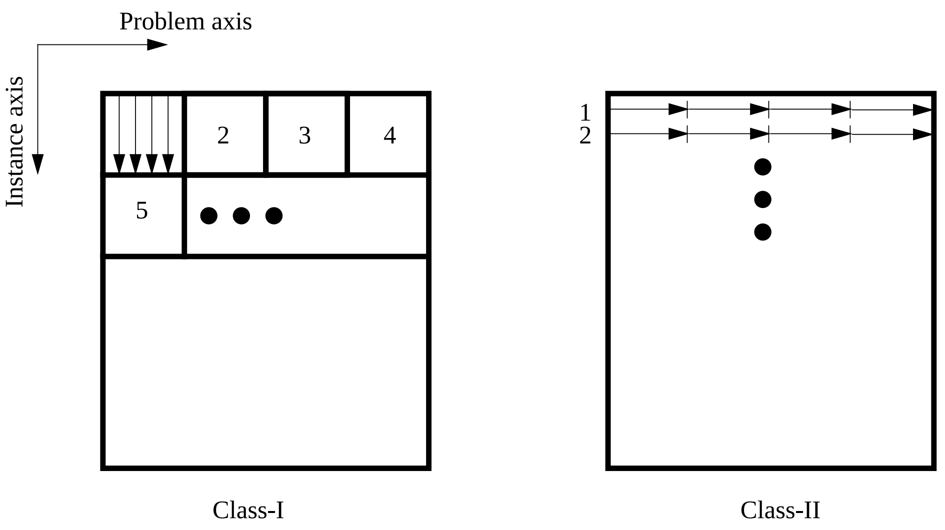


Figure 6: Looping orders of Class–I and Class–II Level–1 LBLAS.

desired. For a DBLAS _NRM2 operation, each instance of C may be distributed across many nodes. The square root operation on the result cannot be performed by each of the participating VUs, i.e, each LBLAS routine. The partial results of each instance of C must first be reduced into a single result, then the square root of the result is taken. _NRM2SQ does compute the square root in parallel on all VUs in addition to the functionality of _NRM2. This case is useful when the user performs a multiple–instance _NRM2, where individual instances are completely local to each VU.

4.2 Optimizations

All Level–1 LBLAS routines have been separately optimized for 32–bit (halfword) and 64–bit (word) data. The primary differences between the two cases arise from the size of the Register File, and from the memory being organized as 64–bit words accessed via a 64–bit data path. Level–1 LBLAS routines limited by the memory bandwidth for input operands (loads) can achieve close to twice the 64–bit performance in 32–bit precision. For example, the peak efficiency for an inner–product on real data in 64–bit precision is 50%, while in 32–bit precision the peak efficiency is 100%. For operations with a relatively large number of output operands, optimization for 32–bit operands may yield a sevenfold speedup. In order to realize this speedup, care has to be taken to ensure that the stored data is aligned with word boundaries.

The loop orderings of the Level–1 LBLAS kernels are divided into two classes. Referring to Figure 6, Class–I kernels are vectorized with the instance–axis innermost. After loading a segment of the instance–axis into memory, the algorithm steps along the problem–axis. Class–II kernels are vectorized with the problem–axis innermost. One entire instance is computed prior to the following instance being addressed.

Before describing the optimization in detail, we summarize the peak performance and half–performance vector lengths achieved for the Level–1 LBLAS on the CM–5E in Tables 4 and 5. The peak performances were achieved using either Class–I or Class–II kernels, whichever gave better performance with its optimal layout. Thus, the layout for Class–I kernels has unit stride along the instance–axes.

Referring to Table 4, the asymptotic efficiency realized by the LBLAS kernels for 64–bit data is seen to be in the range of 85 to 98%. For 32–bit data the efficiency can sometimes be more than doubled by using both operands in a 64–bit load and computing in succession both operands for a 64–bit store (see sections 4.2.1 and 4.2.2).

It is clear from Table 5 that $n_{\frac{1}{2}}$ for 64–bit data de-

Table 4: CM–5E peak measured node performance on Level–1 LBLAS, R_∞.

Function	Peak	Number of instances								
	MFlops/s	1	2	3	4	5	7	10	16	32
DSCAL	40	34.0	34.0	34.0	34.0	34.0	34.0	34.0	33.9	34.0
DAXPY	53.3	47.5	47.5	47.5	47.5	47.5	47.5	47.5	47.4	47.4
DDOT	80	72.6	72.8	73.0	73.4	73.1	73.3	73.1	73.8	74.1
DNRM2	160	154.5	154.7	155.2	155.0	154.7	154.7	153.9	155.4	154.1
SSCAL	53.3	47.5	47.5	47.5	47.5	47.5	47.5	47.3	47.2	47.0
SAXPY	80	73.0	73.0	73.0	73.0	73.0	72.8	72.8	72.6	72.3
SDOT	160	139.0	139.8	140.2	140.3	140.5	139.7	138.6	147.4	145.3
SNRM2	160	155.2	155.4	155.4	155.4	155.4	155.0	155.0	155.7	155.2

Table 5: Level–1 LBLAS CM–5E half–performance vector length $n_{\frac{1}{2}}$.

Function	Mflop/s	Number of instances								
		1	2	3	4	5	7	10	16	32
DSCAL	20.0	7	6	5	3	2	2	4	2	2
DAXPY	26.7	6	4	4	2	2	1	2	1	1
DDOT	40	24	15	10	6	5	3	3	2	2
DNRM2	80	42	36	22	11	9	7	7	5	4
SSCAL	26.6	32	32	32	32	32	32	16	3	2
SAXPY	40.0	33	33	32	33	33	33	33	16	2
SDOT	80	148	102	89	88	88	80	64	12	12
SNRM2	80	59	52	40	16	16	8	6	6	6

creases rapidly as a function of the number of instances. The performance of the node processor is sufficiently high compared to the VUs that the looping overhead is very low in executing the kernels, resulting in high efficiency. The $n_{\frac{1}{2}}$ is either 1 or 2, even for relatively few instances per VU.

The 32–bit data shows a somewhat different pattern. The peak performance for 32–bit data is realized when the stores to memory are performed in 64–bit precision (see Section 2). In order to realize the added performance from this feature, a minimum vector length on the order of the tile size is required (see Table 6). The $n_{\frac{1}{2}}$ remains relatively constant, at the tile length, unless there is a sufficient number of instances to fill a tile along the instance–axis, at which point $n_{\frac{1}{2}}$ drops significantly. The $n_{\frac{1}{2}}$ of the kernels utilizing 32–bit stores is approximately equal to that of the corresponding 64–bit kernels.

The set of tile shapes, vector lengths, and loop unrollings used by the Level–1 LBLAS kernels are summarized in Table 6. As a rule, Class–I kernels are fully unrolled, while Class–II kernels are not necessarily fully unrolled. For fully unrolled kernels the unrolling depth is determined by the vector length being used (maximum 16), and the number of available registers. The tile shapes are specified with the problem–axis length first, and the instance–axis length second.

For all reported timings, the layout of the operands is optimal for the kernel being considered. Thus, for instance, for a Class–I kernel the stride along the instance–axis is assumed to be one, and the stride along the problem–axis is equal to the length of the instance–axis, i.e., M. For Class–II kernels, the stride along the problem–axis is one, and the stride along the instance–axis equal to the length of the problem–axis.

In the plots of kernel performance that follow, there are generally periodic ripples in the performance curves. The ripples are generally a function of the tile shape and

Table 6: The set of tile shapes, vector lengths and loop unrollings used for the Level–1 LBLAS.

Function	Loop Ordering	Tile shapes {P, M}	Vector Length	Unrolling Depth {P, M}
DSCAL	Class–I	{8, 8}	8	{8, 1}
	Class–II	{32, 1}	16	{2, 1}
DAXPY	Class–I	{8, 8}	8	{8, 1}
	Class–II	{32, 1}	16	{2, 1}
DDOT	Class–I	{8, 8}	8	{1, 8}
	Class–I	{16, 4}	16	{1, 4}
	Class–II	{64, 1}	16	{4, 1}
DNRM2	Class–I	{8, 8}	8	{8, 1}
	Class–II	{16, 1}	16	{1, 1}
SSCAL	Class–I	{4, 16}	16	{1, 4}
	Class–II	{64, 1}	16	{4, 1}
SAXPY	Class–I	{4, 16}	16	{4, 1}
	Class–II	{64, 1}	16	{4, 1}
SDOT	Class–I	{8, 16}	16	{16, 1}
	Class–II	{128, 1}	16	{8, 1}
SNRM2	Class–I	{1, 16}	16	{1, 1}
	Class–II	{16, 1}	16	{1, 1}

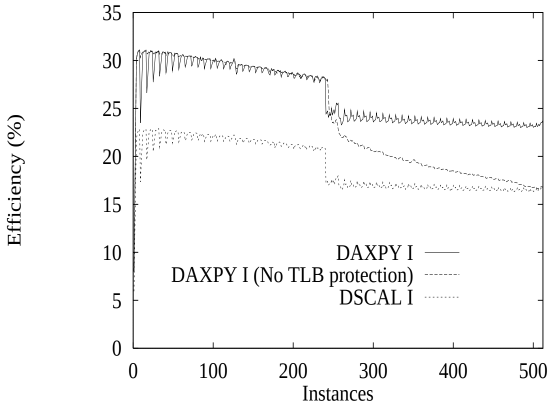
unrolling. There is a distinct performance gain available from decomposing the problem exactly into an integer number of tiles. If this is not possible, then a penalty for under utilizing the tile is accrued. On small problems, under utilizing tiles can significantly affect performance. To minimized this effect, certain kernels alter the tile shape for small problems. For example, consider a problem with an axis length of 9, where the tile extent along that axis is 8. The kernel will decompose the problem into two tiles of size 5 and 4 respectively, rather than tiles of 8 and 1. Due to pipeline startup costs, the minimum instruction execution time is 4 cycles, even for vectors shorter than length 4. Hence, decomposing into tiles of extent 5 and 4 will take $5+4=9$ cycles, while a decomposition into extents of 8 and 1 will take $8+4=12$ cycles. By changing the tile shape, 3 cycles, or 25% of the execution time can be saved.

4.2.1 _SCAL and _AXPY

The _SCAL operation computes $C^T \leftarrow AB^T$, where C and B are two–dimensional arrays (one of which is an instance–axis) and A is a one–dimensional array. The function is performed by loading B into the register file and using a chained vector multiplication. As the CM–5/5E cannot simultaneously support chained stores and loads, the result is stored with a separate vector instruction. Similarly, the _AXPY operation performs the function $C^T \leftarrow C^T + AB^T$.

Figure 7 shows the efficiency as a function of the number of instances for the DSCAL–I and DAXPY–I and DSCAL–II and DAXPY–II kernels. The extent of the problem–axis for the timing of the Class–I kernels is 128, while the extent of the instance–axis for the Class–I kernels is eight. Since DSCAL performs one multiplication per two memory references, the maximum efficiency is 25%. DSCAL–II achieves a peak efficiency of approximately 22% and DSCAL–I a peak of approximately 23%. DAXPY performs a multiply–add operation for every three memory references, at best. The maximum attainable efficiency is 33%. Both the DAXPY–I and DAXPY–II kernels achieve a peak efficiency of approximately 31%.

The maximum vector length used in the Class–I kernels is eight, which is visible as a ripple in the corresponding plot in Figure 7. For the Class–II kernels the maximum vector length of 16 is used together with an



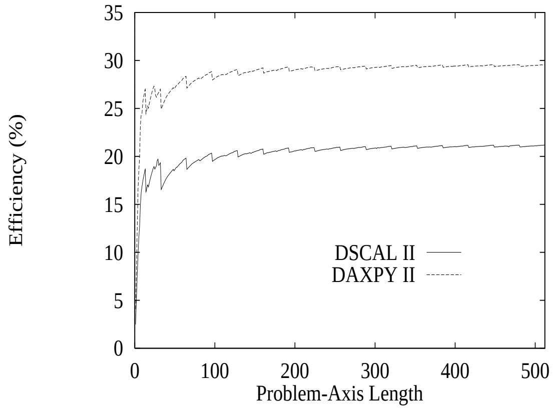


Figure 7: Efficiency of the DSCAL and DAXPY kernels.

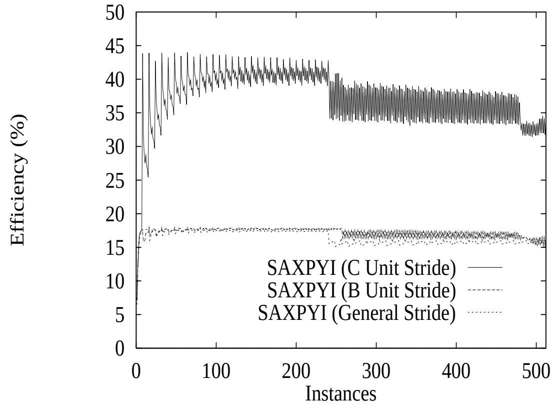
unrolling depth of two, resulting in performance peaks separated by 32 elements.

Due to the loop ordering of the DAXPY–I kernel, it is susceptible to TLB thrashing. This is visible in Figure 7 as a sharp dropoff in performance at 256 instances. An extra loop is placed around the kernel in order to alleviate this problem, as can be seen by the curve for DAXPY–I with TLB protection.

The _SCAL and _AXPY functions are the ones most highly dependent on memory bandwidth of any of the currently implemented LBLAS. The factor of seven speedup available by implementing two 32–bit stores as a single 64–bit store tends to dominate the performance of the 32–bit kernels. Both Class–I and Class–II kernels have been written which exploit this fact. The combination of axis of vectorization and unit–stride results in a large number of kernels. A total of 27 32–bit kernels have been implemented.

The SSCAL kernels have characteristics very similar to the SAXPY kernels. We only discuss the latter here. The SAXPY–I and II kernels have been optimized for unit stride access on either B or C, but not both. The non–optimized kernels for a general (non unit) stride on both B and C achieve a peak efficiency of approximately 16%. The theoretical peak in this case is 18%, because the inner loop of the kernel requires two 32–bit loads (1 cycle each) and one 32–bit store (3.5 cycles) to perform two floating–point operations. A small speedup can be achieved when B has unit stride, since loading an element of B effectively only requires $\frac{1}{2}$ a cycle. A far greater speedup can be achieved when C has unit stride. A load and a store of an element of C each take $\frac{1}{2}$ a cycle, yielding a theoretical maximum efficiency of 50%. A further optimization is possible when both B and C have unit strides. However, this optimization has not been implemented.

The high frequency ripple observed in Figure 8 for the SAXPY–I unit stride kernels is an even/odd stride phenomenon. When the instance stride is one or even and the instances are 64–bit word aligned, then they can be treated as a single block. If the instances are misaligned, then the first element is handled separately, and the rest of the instances are treated as a single block. In contrast, for an odd stride other than one, alternating instances will have alternating alignments (see Figure 9) and we break the problem into three distinct blocks. The first block is the instances that are 64–bit word aligned (half of the instances), which are computed by a single kernel call with a double stride on the problem axis. Next, the first (misaligned) elements of the odd instances are handled with a call to a general–stride, SAXPY–II kernel (invoked with one instance). Finally the rest of the elements of the odd instances are computed by a call to the unit–stride SAXPY–I kernel, with a problem–axis length of one less than the 'true' problem–axis length. In the odd stride cases, the overhead associated with the extra kernel calls, as well as the reduced efficiency of operating on a size 127 rather than 128 problem is apparent in Figure 8. In the case of the C unit–stride kernel this is apparent as an approximately 4% reduc-



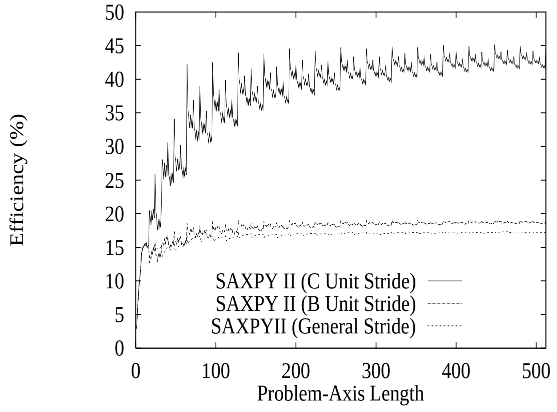


Figure 8: Efficiency of the SAXPY–I and SAXPY–II kernels.

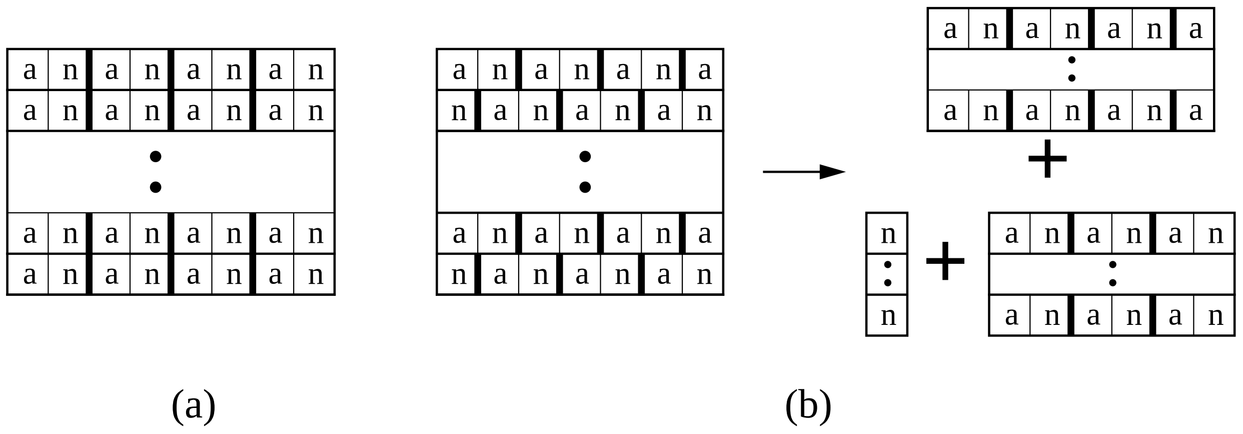


Figure 9: Word alignment of 32–bit data with (a) even and (b) odd (other than one) strides.

tion in efficiency.

For the SAXPY–II kernels in Figure 8, ripple occurs in the performance curves with a frequency of 8, 16 and 32 elements. (This ripple is also present in the SAXPY–I curves, but it is masked by the high–frequency ripple.) The source of the ripple is the high cost of non 64–bit load/stores. Only even, fairly large, blocks of stores can be implemented as 64–bit operations. Eight (32–bit) stores is the minimum number that can be implemented in this way. Blocks of 32 32–bit stores are most efficient (for physical implementation reasons). For the C unit–stride kernel, the cost of 32–bit stores is particularly high, which appears as a ripple with significant amplitude.

Figure 10 shows the performance of the SAXPY kernels for small problem sizes. The general–stride and B unit–stride SAXPY–I kernels have an $n_{\frac{1}{2}}$ of approximately 2. As 64–bit stores cannot be exploited with less than 8 instances, the $n_{\frac{1}{2}}$ of the C unit–stride SAXPY–I kernel is 8. The SAXPY–II kernels have similar characteristics, with the general and B unit–stride kernels having an $n_{\frac{1}{2}}$ of approximately 3, and the C unit–stride kernel an $n_{\frac{1}{2}}$ of approximately 32.

4.2.2 _DOT

The inner–product $C \leftarrow C + AB^T$ reduces the prod-

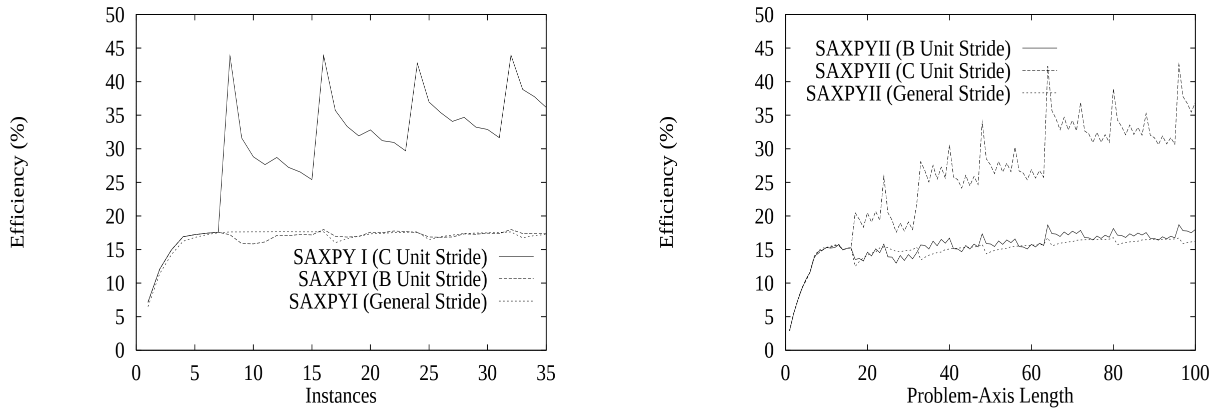


Figure 10: Performance of the SAXPY–I and SAXPY–II kernels on small problems.

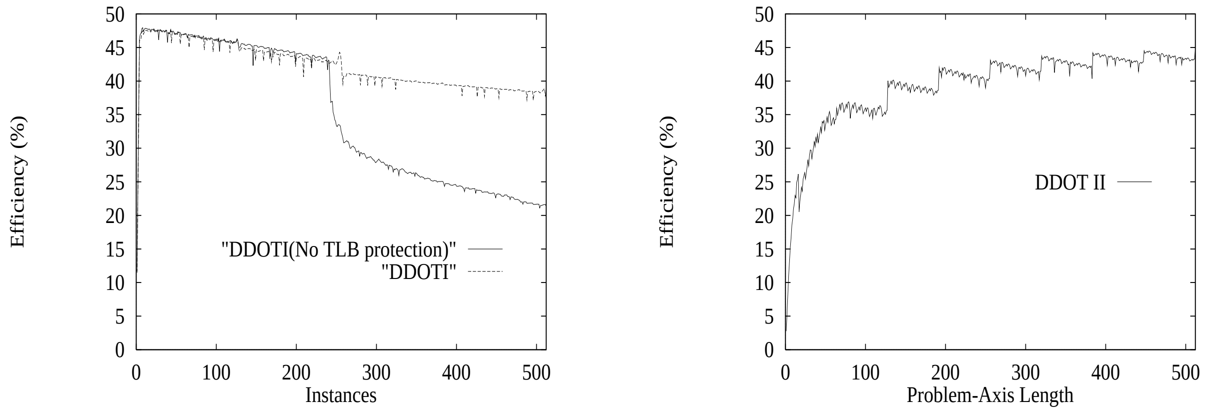


Figure 11: Performance of the DDOT–I and DDOT–II kernels.

uct of two vectors to a single element. In the multiple–instance case, the product of a collection of pairs of vectors reduces to an array of rank one lower than that of the input vectors. Figure 11 shows the performance of the DDOT–I kernel for a problem–axis length of 128. This kernel performs approximately two memory accesses per multiply–add operation, so the efficiency can at best be 50%.

Figure 11 shows the performance of the DDOT–II kernel for eight instances. The kernel uses a maximum vector length of eight with an unrolling depth of eight. The combination of vector length and loop unrolling results in performance peaks for problem–axis lengths being multiples of 64. The performance decreases slowly between multiples of 64 elements due to the increasing significance of lower performing kernels handling the excess elements. The ripple between each 64th element has a frequency of eight. It is due to the efficiency in operating on axes of lengths being integer multiples of the vector length 8. The peak efficiency in the example shown is approximately 46.5%.

Figure 11 shows that the Class–I kernels are very efficient already for very few instances and a problem–axis length of 128. If the operands have both short problem–axis and instance–axis, then Class–I kernels are generally more efficient than Class–II kernels, when there are

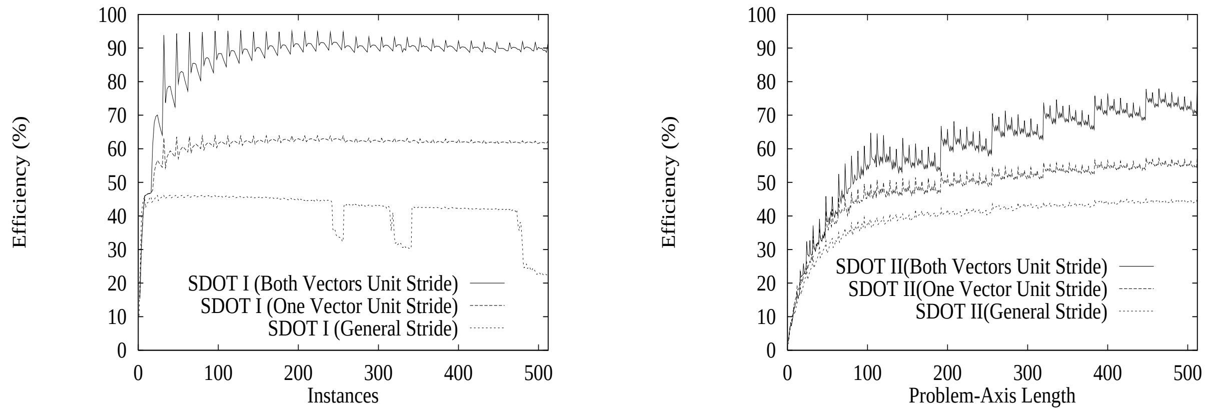


Figure 12: Performance of the SDOT–I and SDOT–II kernels.

four or more instances. For larger problems the layout of the arrays with respect to DRAM page boundaries determines which kernel is more efficient.

Figure 12 shows the efficiencies of the SDOT–I and II kernels. If the vectors of halfword data are laid out in memory with unit stride, then it is possible to use halfword loads and stores. With all operands accessed with unit stride, the measured peak efficiency for the SDOT–I kernels reaches 92% before it drops due to DRAM page faults and TLB entry misses. With an odd axis stride other than one, successive halfwords along an axis alternate between the first and second halves of different full words in memory. The current set of kernels make no attempt to exploit halfword loads and stores for odd strides other than one.

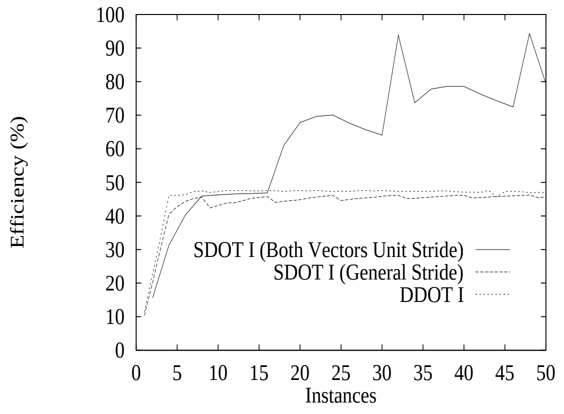
When there are few instances, the SDOT–I kernel (both vectors unit stride) is particularly sensitive to vector lengths that do not fit exactly into the registers. The extra elements are loaded with halfword operations, which is relatively very expensive. This results in the sharp peaks observed in Figure 12.

The performance of the general stride case is similar to that of the DDOT–I kernel and the case with one vector having unit stride falls between the two.

Though only a single element is stored for each inner–product, the relatively high expense for a halfword store makes it important to also address the issue with halfword stores for inner–products on short problem–axis. The need to perform many small inner–products arises in explicit codes for the solution of partial differential equations on a regular grid. For instance, the common five–point centered difference stencil in two dimensions result in inner–products of length five in each grid point. The effect of taking advantage of halfword stores for small inner–products can be significant, however we have not currently implemented this optimization.

In order to support all the modes and precisions, 64–bit alignments, and strides of the three operands, the CMSSL currently includes 51 distinct _DOT kernels. Each kernel tiling is implemented in *add, sub* and *noadd* forms. Thus, the 51 kernels consist of 17 different vectorization schemes. Of these three are 64–bit kernels and 14 are 32–bit kernels. The 32–bit kernels are optimized to take advantage of unit strides along either the problem– or instance–axis. This large number of distinct kernels is necessary because of the possible combinations of 64–bit misalignments associated with performing 64–bit memory operations and 32–bit arithmetic.

The efficiency of the SDOT kernels for small problems is illustrated in Figure 13. The kernels optimized for unit stride requires a vector length of at least 16 in order to exploit the benefits of 64–bit loads, and hence has an $n_{\frac{1}{2}}$ at approximately 16. Local performance maxima occur at vector lengths that are multiples of 16. The Class–II kernels show a far less steep performance curve (due to the overhead associated with reduction along the axis of vectorization). The general stride kernels have an $n_{\frac{1}{2}}$ of approximately 20, and the unit stride kernel has $n_{\frac{1}{2}}$ of approximately 60.



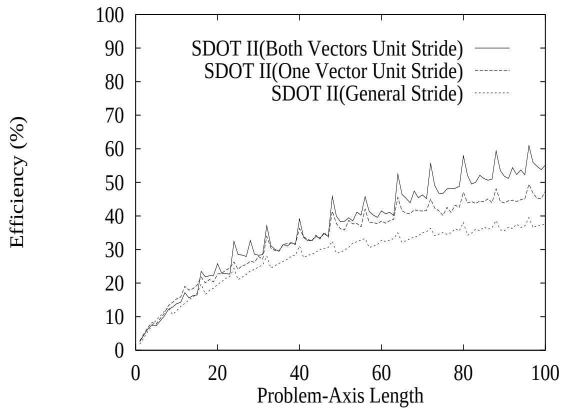


Figure 13: Performance of the SDOT kernels on small problems.

4.2.3 _NRM2, _NRM2SQ and _SIP

The _NRM2, _NRM2SQ and _SIP LBLAS functions all use the same kernels. The only difference being that _NRM2SQ performs a square root operation. For the purposes of reporting the overall performance, this is considered negligible.

The _NRM2 routine performs the operation $C \leftarrow C + AA^T$ and has only one input and one output operand. One memory reference per floating–point multiply–add suffices, giving a theoretical efficiency of close to 100%. Figure 14 shows the performance of the DNRM2 and SNRM2 kernels. The kernel is not generally memory bandwidth limited, so no optimization for word loads, and stores of halfword data has been implemented. The cost of memory operations can become significant if the problem consists of a large number of relatively small instances. At present, no optimizations have been implemented to exploit this situation.

The measured peak efficiency of both the DNRM2 and SNRM2 kernels is approximately 97%. The performance drops off with very large problem sizes, due to DRAM page faults and TLB entry misses, without blocking beyond the register tiles.

Figure 14 shows the performances of the _NRM2–I kernels with a problem–axis of length 128, and the performance of the _NRM2–II kernels for eight instances. With the DNRM2–I kernel the effect of TLB thrashing becomes apparent when the addresses of all of the instances do not fit into the 64 TLB entries. In order to fill the 64 TLB entries, each of which map to 4 Kbytes of memory, approximately $\frac{4\times1024\times64}{128\times8} = 256$ instances are required. In the case of SNRM2–I, the effect is first apparent in the region of 512 instances. In order to avoid TLB thrashing, an extra outer loop has been added to the kernel, such that the inner loop operates only on instances for which the addresses fit into the TLB at one time. The DNRM2–I kernel, for which the performance is shown, also includes this added loop.

Also apparent from Figure 14 is the effect of DRAM page faults and TLB misses on the SNRM2–I kernel. The performance degradation without blocking is smaller than for the DNRM2 kernel due to the smaller data size. For a given stride, page faults are relatively less frequent. The DNRM2–II and SNRM2–II kernels both achieve approximately 99% efficiency for very large problem sizes.

Another significant concern in developing the kernels was low overhead and as smooth a performance behavior as possible as a function of the extents of the problem– and instance–axes. For short axes, the effects of the minimum vector–length–four based instruction set for the VUs, and the node processor overhead for loop control, are significant concerns. Figure 15 illustrates the performance of the kernels for small problems. Figure 15 clearly shows the smoothness of the performance curves in the very short axis region. It is apparent that the $n_{\frac{1}{2}}$ for the Class–I kernels is approximately 3, and that of the Class–II kernels approximately 40. The lower efficiency of the Class–II kernels arises because of the overhead incurred in the reduction along the direction

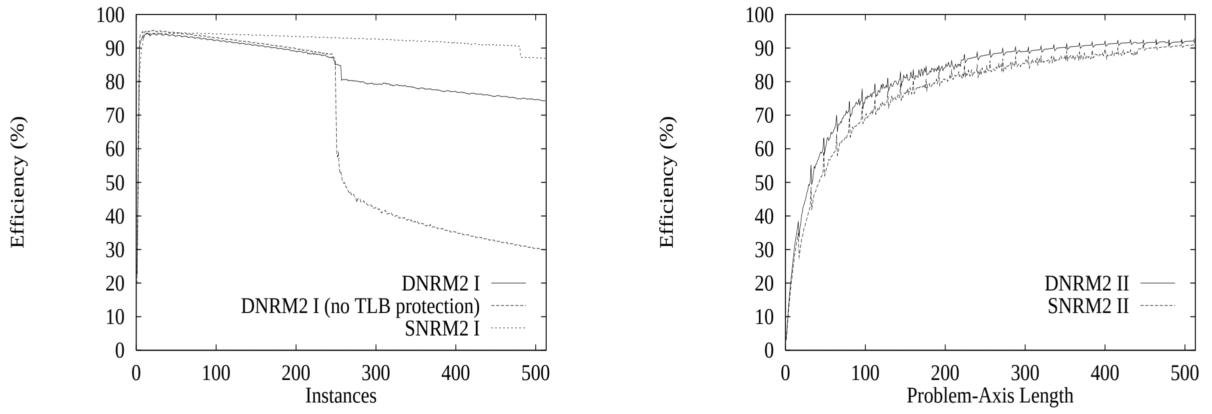


Figure 14: Performance of the DNRM2 and SNRM2 kernels.

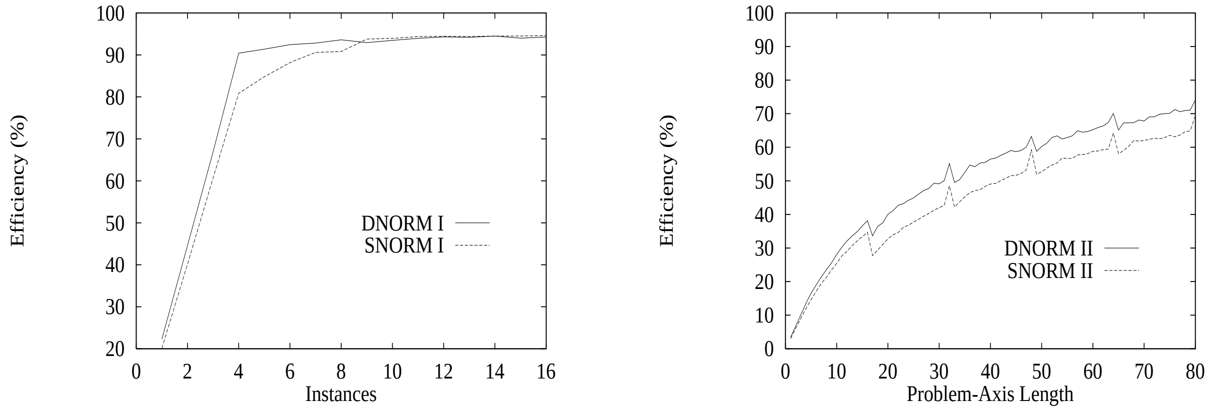


Figure 15: Performance of the DNRM2 and SNRM2 kernels for small problems.

of vectorization.

5 Level–2 LBLAS

5.1 Functionality

Two Level–2 LBLAS functions have been implemented, namely _GEMV and _GER. As for the Level–1 LBLAS, the main difference compared to the standard BLAS is the exclusion of scaling parameters and the inclusion of a multiple–instance capability. The LBLAS interface also supports _GEVM, which is implemented (without performance penalty) as a call to _GEMV, with the strides and axis extents appropriately modified. _GEVM is not discussed further, as it is considered identical to _GEMV for the purposes of this paper.

Table 7 summarizes the peak efficiencies and Table 8 the half–performance vectors for the Level–2 LBLAS. The half–performance figure is computed for square matrices. Similar to the peak performance tables for the Level–1 LBLAS, the peak performances in Table 7 and 8 were achieved using the loop ordering that gave the best performance (see next section) with its optimal array layout.

Table 7: Peak efficiencies for Level–2 LBLAS with square matrices.

Function	Peak	Number of instances								
	MFlops/s	1	2	3	4	5	7	10	16	32
DGER	80	69.9	70.2	70.4	70.4	70.4	69.4	69.8	69.6	68.8
DGEMV	160	140.0	140.3	139.8	140	140.0	139.4	136.3	135.5	135.8
SGER	160	122.9	122.9	122.9	122.7	122.4	122.4	122.4	121.1	121.0
SGEMV	160	149.6	147.4	146.7	146.9	146.7	146.9	147.0	146.9	147.0

Table 8: Half–performance axis length for Level–2 LBLAS with square matrices.

Function	MFlops/s	Number of instances								
		1	2	3	4	5	7	10	16	32
DGER	40	6	6	5	4	4	4	4	4	4
DGEMV	80	10	8	8	6	6	6	8	8	8
SGER	80	32	32	32	32	32	32	32	32	32
SGEMV	80	9	8	8	8	8	8	8	8	8

5.2 Optimizations

For the Level–2 LBLAS three–dimensional register tiles are used for some loop orders. The tile shapes, vector lengths and loop unrollings used for the Level–2 LBLAS are summarized in Table 9.

5.2.1 _GER

The LBLAS outer–product operation is performed on two two–dimensional arrays A and B^T, yielding a three–dimensional array C of shape $P \times R \times M$. The number of instances is M, as before. The input operands each contain one problem–axis, while the output operand has two problem–axis. As for other LBLAS functions, all operands have one instance–axis. The loop orderings available to outer–product kernels can be discussed in terms of the axes labeling of the destination operand C. Some potential loop orderings are shown in Figure 16. In discussing loop orderings we refer to an axis by its extent, and the loop ordering as a sequence of axes extents with the order from left to right corresponding to loops from the innermost to the outermost. Thus, $PRPRM$ refers to a loop order in which elements within a column is accessed in the innermost loop, the vectorized loop, and columns accessed in the next outer loop. The two innermost loops together form the register tile. The next outer loop handles all tiles along a column, followed by a loop for all tiles along a row. Multiple–instances are handled in the outermost loop. Reducing the number of DRAM page faults and TLB misses may require additional loop partitioning resulting in up to nine letter strings defining the loop order.

For all reported timings for Level–2 LBLAS, the layout of the operands is optimal for the kernel being considered, i.e., the strides of the axes are increasing from the innermost to the outermost loop. Thus, for instance, for loop ordering $PRPRM$, the P–axis has unit stride, the stride along the R–axis equals the extent of the P–axis, and the stride of along the M–axis equals the product of the extents of the P and the R–axes.

The outer–product requires that tiles of A and B be loaded into the Register File. The loading of C is chained with the multiplication and addition of the appropriate pieces of A and B. The tile of C is then stored. One of the tiles of A or B can be kept in registers, while the next tile of that operand is loaded. The process is repeated until one of the vectors is exhausted. Then, the next tile of the other vector is loaded. Given a sufficiently large Register File, approximately $M(PR + P + R)$ loads and MPR stores are required, while $2MPR$ floating–point operations are performed, leading to a maximum efficiency of approximately $2MPR/(2 \times 2MPR) = 50\%$.

Kernels which utilize loop orderings $PRPRM$,

Table 9: The set of tile shapes, vector lengths and loop unrollings used for the Level–2 LBLAS.

Function	Loop Ordering	Tile shapes {P, Q, M}	Vector Length	Unrolling Depth {P, Q, M}
DGER	PRPRM	{16, 16, 1}	16	{1, 16, 1}
	PRRPM	{16, 16, 1}	16	{1, 16, 1}
	RPRPM	{16, 16, 1}	16	{16, 1, 1}
	RPPRM	{16, 16, 1}	16	{16, 1, 1}
	MPRPRM	{2, 4, 8}	8	{2, 4, 1}
	MPRRPM	{4, 2, 8}	8	{4, 2, 1}
DGEMV	PQPQM	{16, 32, 1}	16	{1, 32, 1}
	PQQPM	{32, 16, 1}	16	{32, 16, 1}
	QPQPM	{4, 8, 1}	8	{4, 1, 1}
	QPQPM	{8, 4, 1}	4	{8, 1, 1}
	MQPQPM	{2, 4, 8}	8	{2, 4, 1}
	MPQQPM	{4, 2, 8}	8	{4, 2, 1}
SGER	PQPQM	{16, 16, 1}	16	{1, 16, 1}
	PQQPM	{32, 32, 1}	16	{2, 64, 1}
	MPQPQM	{8, 4, 8}	8	{8, 4, 1}
	MPQPQM	{2, 4, 16}	8	{2, 4, 1}
	MPQQPM	{8, 4, 8}	8	{8, 4, 1}
	MPQQPM	{4, 2, 16}	16	{4, 2, 1}
SGEMV	PQQPM	{64, 32, 1}	16	{4, 32, 1}
	PQPQM	{32, 64, 1}	16	{2, 64, 1}
	QPQPM	{8, 8, 1}	8	{8, 1, 1}
	MPQQPM	{4, 3, 16}	16	{4, 3, 1}
	MPQQPM	{8, 7, 8}	8	{8, 7, 1}
	MQPQPM	{3, 4, 16}	16	{3, 4, 1}
	MQPQPM	{7, 8, 8}	8	{7, 8, 1}

$PRRPM$, $MPRRPM$ and $MPRPRM$ have been implemented. By manipulating the strides and axes order in the routines' argument list, the kernels can emulate the loop orderings $RPRPM$, $RPPRM$, $MRPPRM$ and $MRPRPM$, respectively. The remaining loop orderings either can be shown to yield lower efficiency than these orderings, or when implemented were found to yield no significant advantage. The choice of which kernel to invoke is made at run–time and is dependent on the strides and lengths of the result matrix C (see section 7).

Generally, kernels with P–axis vectorization shall be used when the P–axis of C has smallest stride. If the R–axis has smallest stride, then the same kernels are used with the A and B operand arguments switched. For example, the $PRPRM$ kernel then effectively becomes a $RPRPM$ kernel.

Figure 17 illustrates the performance of the $PRPRM$ and $PRRPM$ kernels with eight instances. The minimum number of memory cycles for the $PRPRM$ kernel is approximately $2PRM+RM$ and that of the $PRRPM$ kernel is approximately $2PRM+PM$. For oddly shaped matrices, it is beneficial to use the $PRRPM$ kernel when $R \gg P$. In all other cases, the $PRPRM$ kernel is marginally to significantly faster, because it incurs fewer page faults. Also, the $PRPRM$ kernel does not suffer from the TLB thrashing problem, due to the direction of vectorization. Thus, in most circumstances, the $PRPRM$ kernel is preferred over the $PRRPM$ kernel.

Figure 18 illustrates the performance of the $PRPRM$ kernel for small problem sizes. The half–performance vector lengths for this kernel are seen to be in the range

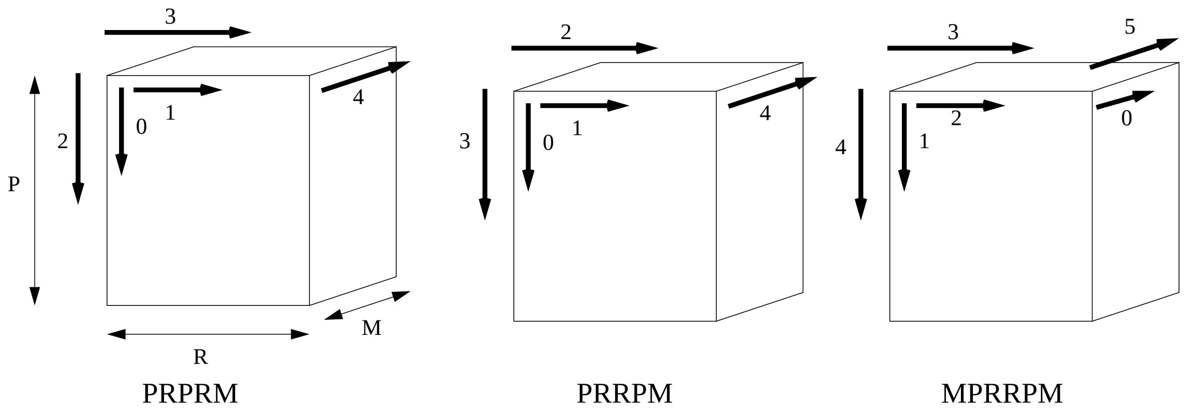


Figure 16: Loop orderings for _GER.

of 2 to 4.

When problem–axes are short, or the instance–axis is the innermost axis of C, then vectorization along the instance–axis often yield the best performance. Thus, kernels with the loop orderings $MPRPRM$ and $MPRRPM$ have been implemented. The performance of these kernels is shown in Figure 19. It is apparent from this Figure that the peak performance of the instance–axis vectorized kernels is significantly less than that of the row–axis vectorized kernels (Figure 17). The reason for this disparity is that the instance–axis vectorized kernels require two– and three–dimensional tiles to be stored in the Register File. The problem–axis vectorized kernels require only one– and two–dimensional tiles. For example, the $PRPRM$ kernel has a tile with P– and R–axis lengths 16, but the $MPRPRM$ kernel has a tile with extents 8, 4 and 2 along the M–, P– and R–axis, respectively. Three dimensional tiles have small extents resulting in the reduction in efficiency evident in Figure 19. The instance–axis kernels also suffer from TLB thrashing. A loop–nest has been added in order to minimize the impact of this phenomenon.

The looping orders that have been implemented for the 32–bit _GER kernels are similar to those for the 64–bit kernels. Performance of the SGER kernels is illustrated in Figure 20. Using only 32–bit loads and stores, the asymptotic performance level of the kernels is somewhat less than two floating–point operations for each 4.5 memory cycles (due to the 3.5 cycle cost of a 32–bit store), or an efficiency of 22% (not counting the cycles for loading A and B etc.)

The performance of the _GER kernels is dominated by the time to load and store C. The 32–bit kernels have been optimized to exploit 64–bit memory operations on C. The efficiency of thusly optimized versions of the $PRPRM$ and $MPRPRM$ kernels is shown in Figure 21. The asymptotic efficiency of these kernels is 100%.

The effects of blocking are particularly apparent in the performance curves of the kernels optimized for unit–strides. This is due to the relatively large overhead of the non unit–stride stores which must be performed on the tail–end of the axis when the axis length is not evenly divisible by the block size (in Figure 21 the block size is 16). As the size of the operands increases, the relative cost of these additional stores diminishes, so the amplitude of the ripples decreases as axis–length increases.

5.2.2 _GEMV

There are 36 choices for loop orderings of matrix–vector multiplication, $C \leftarrow AB$, where A is of shape $P \times Q$, B is $Q \times 1$ and C $P \times 1$. One of the interesting loop orderings for matrix–vector multiplication corresponds to the Level–1 LBLAS routine _AXPY; a sequence of operations $Z(:) \leftarrow X(i)A(:,i) + Z(:)$ are performed. But, in _GEMV the accumulation vector $Z(:)$ is allocated to the Register File and elements of the vector X are preloaded. The columns $A(:,i)$ are loaded from memory as needed.

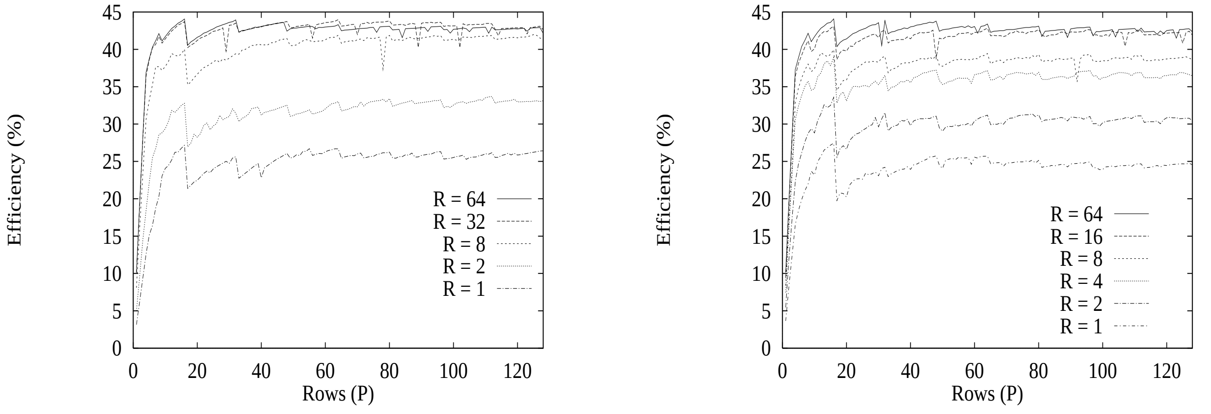


Figure 17: Performance of the $PRPRM$ and $PRRPM$ DGER kernels.

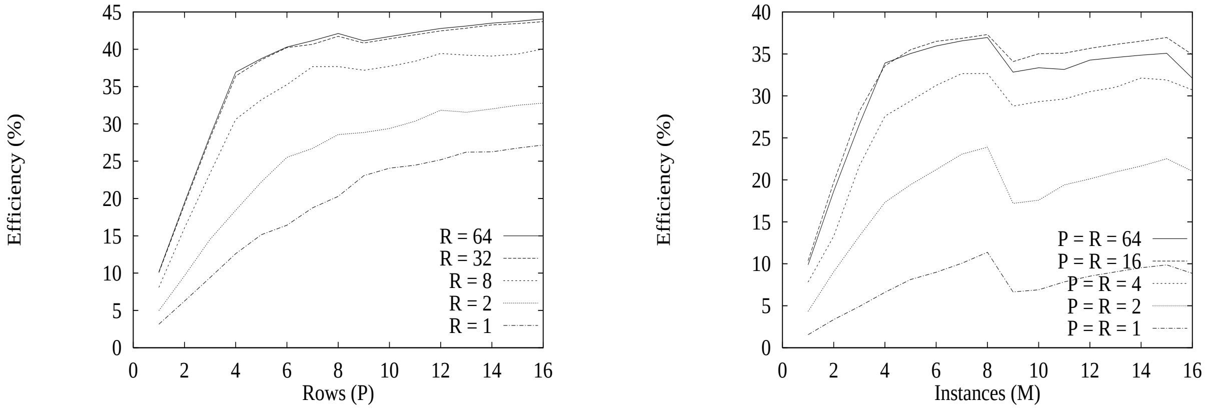


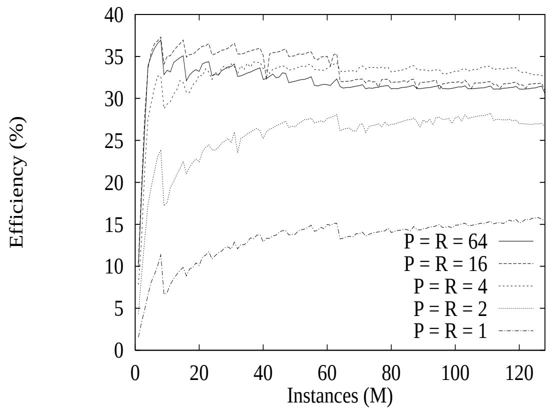
Figure 18: Performance of the $PRPRM$ and $MPRPRM$ DGER kernels on small problem sizes.

The order of the two innermost loops is PQ. Using this loop order, operations on several columns are chained together.

Interchanging the P and Q loops, results in a matrix–vector multiplication based upon inner–products. But, used in the context of _GEMV, one of the input operands, x, is shared between all instances of the other operand. Thus, the Level–1 LBLAS routine _DOT is not used for matrix–vector multiplication. For the QP loop ordering vectorization is on the Q–axis, but the reduction is performed on several rows simultaneously. The two loop orderings of interest in treating a succession of tiles are PQ and QP, creating loop orderings $QPPQ$ and $QPQP$, respectively. With a vector length of VL along the Q–axis, every VLth element along this axis is added together. A final reduction on VL partial sums is required.

For matrix–vector multiplication with multiple–instances, one can simply put the instance–axis outermost. But, for large strides along the problem–axes, vectorizing along the instance–axis (Figure 22) may yield better performance even though no reduction takes place along the M–axis. This involves the sequence of operations $z(i,:) \longleftarrow x(j,:)A(i,j,:)+z(i,:)$ (with the freedom of switching the i and j loops).

Interesting loop orderings without a need to consider DRAM page faults and TLB misses are any ordering of {P,Q,M} within a register tile (called inner loops) fol-



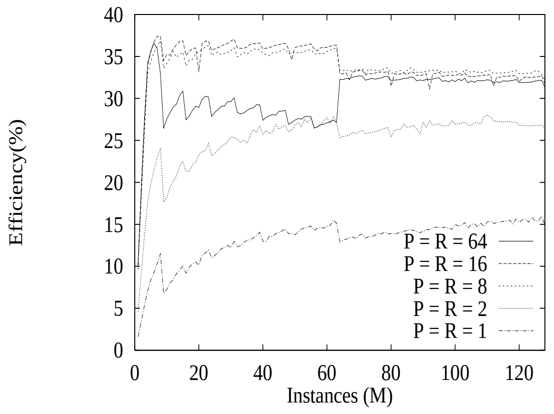
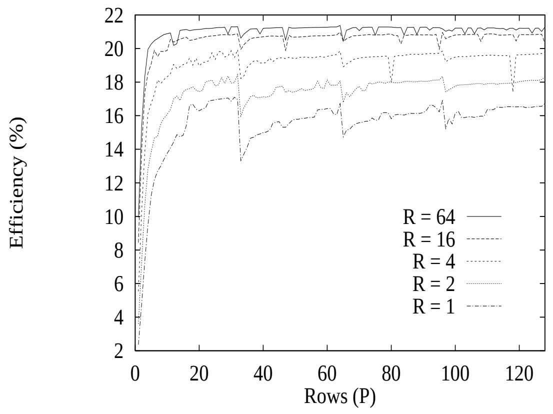


Figure 19: Performance of the $MPRPRM$ and $MPRRPM$ DGER kernels on square matrices.



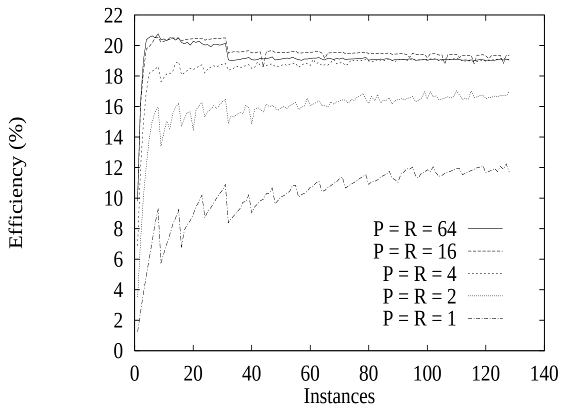
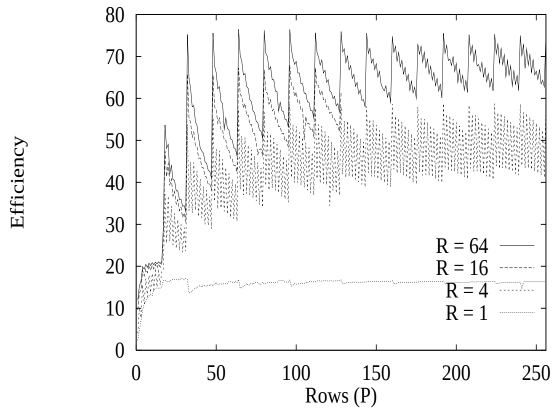


Figure 20: Performance of the $PRPRM$ and $MPRPRM$ SGER kernels.

lowed by any ordering of {P,Q,M} for looping over tiles (called outer loops). Fortunately, five loop orderings (the ones marked with asterisks in Tables 11 and 12) suffice to guarantee optimal or close to optimal performance for all shapes and allocations of the operands A, B, and C. These loop orderings are: $PQPQM$ (AXPY like), $PQQPM$ (AXPY like), $QPQPM$ (inner–product like), $MPQQPM$ (instance–axis vectorization) and $MQPQPM$ (instance–axis vectorization).

Figure 23 shows the performance of the kernels with loop orderings $PQPQM$ and $PQQPM$. The register tile shapes for the two kernels are $\{\alpha, \beta, \gamma\} = \{16, 32, 1\}$ and $\{32, 16, 1\}$, respectively, where $\{\alpha, \beta, \gamma\}$ are the extents of the tile in the P, Q and M dimensions, respectively. Because these kernels vectorize along the P–axis of A, the P–axis has been chosen as the independent variable in Figure 23. The load/store time complexity (ignoring DRAM page faults etc.) of the $PQPQM$ loop order is $(2P + PQ + \frac{QP}{\alpha})$, and that of the $PQQPM$ loop order is $(Q + PQ + \frac{2PQ}{\beta})$. When both P and Q are large, both of these are dominated by the PQ term and hence their performances are approximately equal. With unit stride on the P–axis, the $PQPQM$ order incurs less page faults and, in general, is faster than the $PQQPM$ order, particularly when P is large. Therefore, the $PQQPM$ order is used fairly rarely, and the $PQPQM$ order is used when the stride along the P–axis is small or the extent of this axis is large.



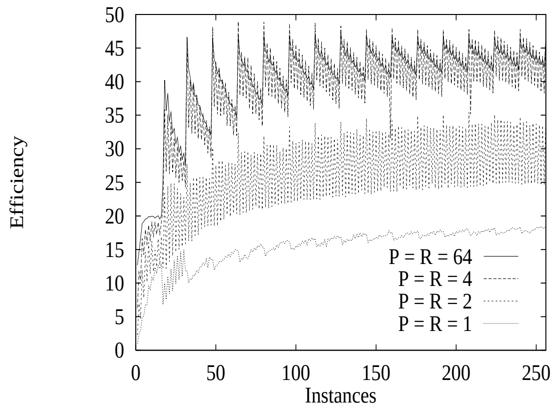


Figure 21: Perfo it–stride matrices.

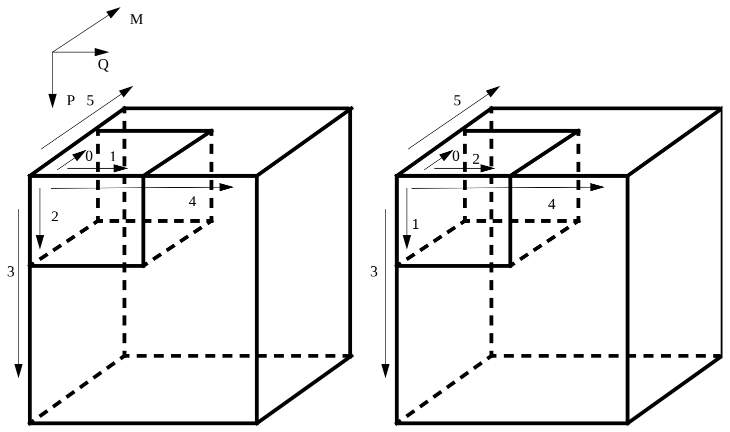
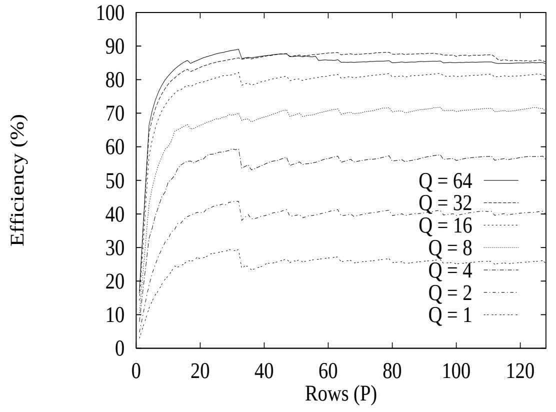


Figure 22: Loop orderings $MQPPQM$ and $MPQPQM$ for matrix–vector multiplication.

Vectorizing along the P–axis is generally the most efficient of the three options. Kernels that vectorize along the Q–axis require a reduction in the direction of vectorization. This reduction can result in a significant decrease in efficiency. Figure 24 shows the performance of the kernel with the loop order $QPQPM$. In Figure 24, the number of columns is chosen as the independent axis, because the kernel vectorizes along this axis. The peak performance of the $QPQPM$ loop order is significantly worse than that of the loop order $PQPQM$. However, when the stride along the Q–axis is small and the stride along the P–axis is large, the $QPQPM$ loop order still performs better than the $PQPQM$ loop order. This fact is illustrated in Figure 24, where A has unit stride along the Q–axis. When the Q–axis is innermost and has a large extent, the cost of page faults reduces the efficiency of the $PQPQM$ kernel, even for large values of P.

If A has been laid out with the instance–axis innermost, or with a relatively long instance–axis, it is often beneficial to vectorize over the instance–axis. However, because instance–axis vectorized kernels require a larger register set than problem–axis vectorized kernels, their efficiency is significantly lower. Two instance–axis vectorized _GEMV loop orders have been included in the CMSSL. These are $MPQQPM$ and $MQPQPM$.



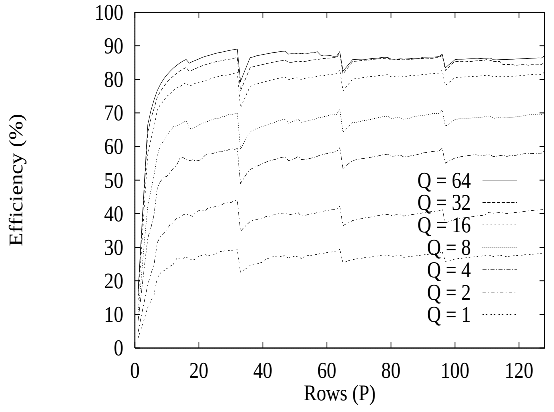
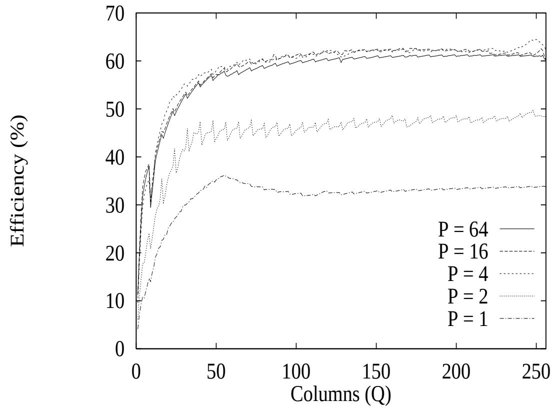


Figure 23: Performance of the $PQPQM$ and $PQQPM$ DGEMV kernels.



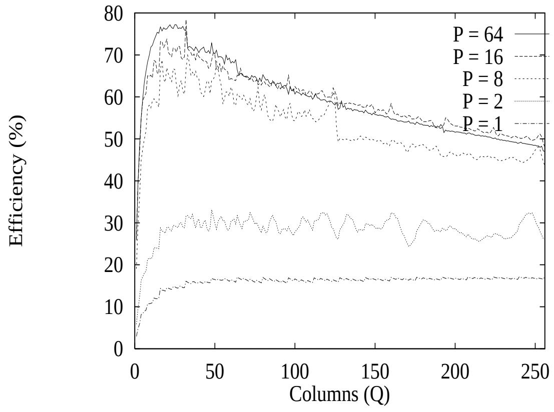


Figure 24: Performance of the $QPQPM$ and $PQPQM$ DGEMV kernels with the Q–axis having stride one.

The register tile that has been implemented is $\{8, 4, 2\}$. Several other register tile shapes were investigated, but none were found to offer significant benefit. The two chosen kernels are identical with respect to register usage, but the inner looping order can be adapted to the layout of A. For this reason, only the performance of the $MPQQPM$ kernel is discussed. The performance of this kernel on square matrices is shown in Figure 25. The peak performance is significantly worse than that of the $PQPQM$ kernel when the stride along the P–axis is small. Thus, the $MPQQPM$ kernel is only used when a significant number of page faults would be incurred vectorizing along the P–axis. These kernels have a small value of $n_{\frac{1}{2}}$, on the order of 2 or 3, (this can be read off of the right–hand set of curves Figure 25). The instance–axis vectorized kernels suffer from TLB thrashing, and an extra level of looping has been added to minimize this effect.

The only difference in the current LBLAS between 64–bit and 32–bit _GEMV kernels is that the 32–bit kernels use twice as many registers. This results in a greater flexibility in the selection of $\{\alpha, \beta, \gamma\}$. In 32–bits, the $PQPQM$ kernel has the register tile $\{32, 64, 1\}$, the $PQQPM$ kernel has a $\{32, 16, 1\}$ register tile, the $QPQPM$ kernel has a $\{8, 8, 1\}$ tile, and the $MPQQPM$ and $MQPQPM$ kernels have a $\{8, 8, 7\}$ tile. The performance of these kernels is similar to that of the corresponding 64–bit kernels.

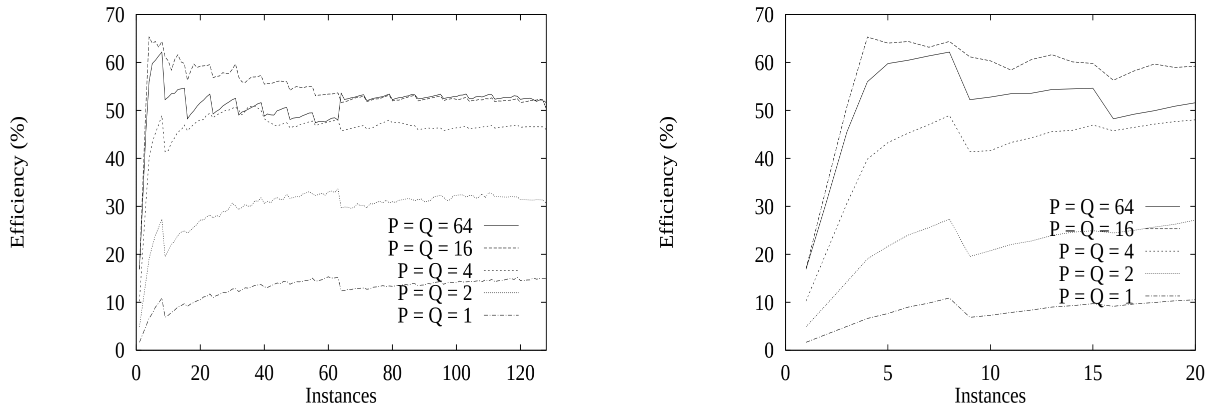


Figure 25: Performance of the $MPQQPM$ kernel on square matrices.

6 Level–3 LBLAS

The memory architecture of the CM–5/5E has some of the features that prompted the design of Level–2 and Level–3 BLAS. The primary motivation for the design of the Level–3 BLAS was the appearance of cache–based memory systems. By appropriately blocking the access pattern of the memory references, the application can exploit the speed of the cache. The CM–5/5E vector units do not utilize a data cache. However, there is a need for blocking in order to minimize DRAM page faults and TLB thrashing. This blocking is already built into our Level–2 LBLAS. The Level–2 _GEMV kernels are balanced between arithmetic load and memory load for a single data path to memory, except for the vector loads and stores. Thus, on the CM–5/5E, only a relatively small gain can be expected from a Level–3 LBLAS for most matrix shapes. We estimated the gain based on memory bandwidth requirements for many matrix shapes to 10 – 15%.

Though the estimated gain for Level–3 LBLAS kernels was small, we implemented several DGEMM kernels using Level–3 blockings. But, the performance of these kernels never exceeded that of the corresponding Level–2 kernels, and was often significantly worse. The reason for this result is that Level–3 kernels require a larger working data set to be held in the registers. This necessitates that the lengths of individual vector instructions be short, which in turn may increase memory traffic and overhead. Thus, no Level–3 LBLAS is included in the CMSSL and the LBLAS _GEMM is implemented by calls to Level–2 kernels.

_GEMM for real data is performed by a sequence of calls to a single kernel. One of the matrices is decomposed into a set of vectors, and the operation is performed as a sequence of either matrix–vector, or vector–matrix multiplications.

Complex multiplication is often decomposed as four real multiplications and four real additions. Similarly for complex matrix–matrix multiplication, which results in a complexity of $\{4\gamma n^3 +$ *lower order terms*$\}$ on matrices of size $n \times n$. For complex operands, the $M3$ method (Hingham, 1992) makes use of a decomposition of complex multiplication into three real multiplications and five additions, resulting in an overall complexity of $\{3\gamma n^3 +$ *lower order terms*$\}$. In the case of the CMSSL implementation of _GEMM, the $M3$ method leads to a speedup of 10 – 15% over the four multiplication method, for sufficiently large matrices. When the matrices are small, it is not beneficial to use the $M3$ method and the traditional approach is retained.

7 Kernel selection

The kernel invocation is a run–time decision that may have significant performance impact. Often entire applications are heavily dependent on one or more LBLAS calls inside an inner loop. The difference in performance of two kernels on a given layout can be as much as an order of magnitude. It is therefore important to give every consideration as to how the kernels are se-

Table 10: Kernels available to selection mechanisms.

Operation	Kernel Options
Level–1	
DSCAL and DAXPY	Class–I, Class–I (with TLB protection), Class–Π
SCAL and SAXPY	Class–I: General stride, C unit stride, B unit stride Class–Π: General stride, C unit stride, B unit stride
DDOT	Class–I: {1,8,8} {1,16,4} Class–Π
SDOT	Class–I: General Stride, Unit stride (9 choices depending on layout) Class–Π: General Stride, Unit stride on one operand, Class–Π: Unit stride on both operands
_SIP	as with _NRM2
DNRM2	Class–I, Class–Π
SNRM2	Class–I, Class–Π
_NRM2SQ	as with _NRM2
Level–2	
DGER	$PRPR$, $PRRP$, $MPRPRM$, $MPRRPM$
SGER	Each with general or unit stride: $PRPR$, $PRRP$ $MPRPRM$, $MPRRPM$
DGEMV	$PQPQM$, $PQQPM$, $QPQPM$, $MPQQPM$, $MQPQPM$
SGEMV	$PQPQM$, $PQQPM$, $QPQPM$, $MPQQPM$, $MQPQPM$

lected. On the other hand, kernel calls for small problems execute very rapidly, and any overhead incurred in deciding which kernel to invoke can result in a significant degradation in performance. Several strategies have been adopted to minimize the cost of kernel selection. Firstly, a low–overhead interface to the LBLAS has been implemented. This interface utilizes a set of cached parameters, whereby only the parameters that change between calls need to be changed when invoking the LBLAS. Secondly, the kernel selection mechanisms have a mechanism for determining that a small problem is at hand and that the overhead for selecting the optimal kernel may well be worse than that of an incorrect kernel selection. In this case, a less rigorous but quick selection is made. Furthermore, the kernel selection mechanism is only invoked if certain parameters, such as axis extents, change. Therefore, if the kernel call is embedded in a loop where, for instance, only the addresses of the operands changes, the kernel selection is bypassed after the first iteration. When kernel selection has been done, the addresses of all of the appropriate *add*, *sub* and *noadd* kernels (see Section 4.2.2) are stored and each can be repeatedly invoked. This is useful, for example, when executing successive kernel calls in order to implement complex arithmetic. Only a single kernel selection is made, while four kernel calls are invoked.

If a 'large' problem is to be executed, it is important that the kernel that is selected is close to optimum in performance. Table 10 shows the choices that the selection mechanism must make for the available LBLAS. The key factors in determining this selection are the extents and strides of the axes of the operands. Several mechanisms were tested in order to determine which provided the most accurate and fastest evaluation. In almost all cases it was found that the quickest and most accurate method was to determine the order of the axes in increasing stride. If the axis with the smallest stride has a sufficiently long extent (typically on the order of 8 to 16), then the kernel that vectorizes over this axis is selected. When more than one kernel exists with the same vectorization (e.g. DDOT and DGER), then it is necessary to examine the other characteristics of the arrays. The kernel selection of the Level–2 LBLAS is typically based on the operand that is a three–dimensional array, rather than those that are two–dimensional, because the three–dimensional array typically dominates the execution time.

8 Conclusions

This paper discusses the implementation and performance details of the LBLAS for the CM–5/5E. The LBLAS are implemented in *SPMD* mode and optimized for the vector units of the CM–5/5E. In order to achieve a maximum degree of freedom in exploiting parallelism, the CMSSL allows for *multiple–instance* function calls. To accommodate this, the LBLAS have been optimized for either problem–axis or instance–axis vectorization. To ensure as robust and uniformly smooth performance as possible, a run–time selection mechanism is invoked which chooses amongst the available kernels. This selection mechanism must be both fast and accurate, in order to ensure high performance for small as well as large problems. Several mechanisms have been implemented to improve performance on problems that are small, or have unusual aspect ratios. These include a low weight calling sequence, a high speed path through the selection mechanism and careful consideration of the vectorization and loop unrolling within the kernels.

The LBLAS supports the data parallel languages CMF and C* used either in global mode, or in local mode with message passing, or in the mixed mode global/local corresponding to HPF with extrinsic procedures. To implement BLAS within such an environment, we have implemented a two–level structure. The DBLAS perform any necessary data motion, and some reduction. The LBLAS perform the actual computation. The performance optimization efforts performed at the LBLAS level can be exploited both locally as well as globally. The LBLAS interfaces have been designed to transparently enable both local and global calls.

The latest version of the CMSSL contains 238 distinct LBLAS kernels, totalling approximately 200 000 lines of C and assembly code. A wide variety of additional codes were analyzed or written and tested to evaluate for potential inclusion. The result is a uniformly high–performance set of codes which have found widespread implementation in user codes.

Appendix: The time complexity for matrix–vector loop unrollings

The time complexity of an operation depends on the CPU utilization and memory traffic (both bus cycles and page fault overheads). Tables 11 and 12 list a number of loop orders, and their time complexity. Those marked with an asterisk (*) in Tables 11 and 12 have been chosen for inclusion in the CMSSL.

- For outer loops, only the orderings {PQM,QPM} are of interest, since if M is made an inner loop, either B or C or both are loaded for every iteration of the outer loops, and never get reused. These cut the ordering choices down to 12, as listed in Tables 11 and 12. Tables 11 and 12 also list the load and store cycles needed for 64–bit kernels, the optimal tile extents $\{\alpha, \beta, \gamma\}$ for 64 registers, and the number of page faults. We give exact formulas for the first four orderings while leaving out the low order terms for the other orderings.

- The loop orderings $QPPQM$ and $QMPPQM$ need $P \times Q \times M$ cycles for the reduction of the running sum vector to one element, and is slower than other orderings most of the time.

- The orderings $PMQPQM$ and $PQPQM$ both are good for small strides on the P–axis. Making the P–axis innermost of the three outer loops results in relatively few page faults. When the M–axis has the second smallest stride, then the ordering $PMQPQM$ has a better chance of incurring the fewest page faults. Consider as an example the tiles $\{32, 16, 1\}$ for the loop order $PQPQM$, and tiles with shape $\{8, 2, 4\}$, or $\{8, 4, 2\}$ for the loop order $PMQPQM$. The two orderings will have roughly the same number of page faults because of the longer loop extents of P in the $PQPQM$ ordering. For this reason, loop orderings $PMQPQM$, $PMQQPM$, and $QMPQPM$ are not implemented. The orderings $MPQPQM$ and $MPQQPM$ will give similar performance as the orderings $MQPPQM$ and $MQPQPM$ when M has the smallest stride, but will lose to some other orderings on other layouts, and are not implemented.

- There are two interesting choices for the outer loops, namely PQM and QPM as explained above. No performance benefit can be derived from having the M–axis loop anything but outermost in the outermost loop nest. The outer loop ordering determines the load/store cycles of B and C. For the ordering PQM, elements of B are reused for whole columns of A and the total number of loads is $Q \times M$. Elements of C are loaded and stored for every iteration of the outer loops, and the total is $2 \times P \times M \times \lceil \frac{Q}{b} \rceil$. Similarly, the loads/store cycles for the QPM outer loop ordering is $Q \times M \times \lceil \frac{P}{a} \rceil$ for B and $2 \times P \times M$ for C.

  Since the shape of the inner loop region is roughly a square, choosing the outer loop ordering is mainly based on the extents of the P and Q–axes. The rule of thumb is to put the shorter axis outside the longer one. The difference will become apparent only when P and Q are very different.

- Any loop ordering with the Q–axis being the inner most is based on inner–product formulations of the matrix–vector multiplication, and thus needs "cleanup" for vectorized inner–products. Examples of such loop orderings are $QPPQM$, $QPQPM$, $QMPPQM$, and $QMPQPM$.

- In the formulas for the number of page faults for each ordering, the highest order terms are always on the innermost axis – the axis being vectorized. The rule of thumb in picking optimal kernels is always to pick the ordering that puts the axis with smallest stride innermost.

Biographies

David Kramer

David Kramer received the B.Sc (Eng) degree with distinction from the University of the Witwatersrand, South Africa in 1987. He received the M.A. and Ph.D. in Computer Engineering from Princeton University in 1990 and 1993 respectively. From 1992 through 1995 Dr. Kramer was a scientist at Thinking Machines Corporation, where he participated in the development of the CMSSL, run-time systems and high performance message passing libraries. Since October 1995, Dr. Kramer has been with the Oracle Corporation where his focus is on commercial applications of high performance computing systems. This work was conducted when author was with Thinking Machines Corporation, Cambridge, Massachusetts.

S. Lennart Johnsson

Dr. S. L. Johnsson received the M.S. and Ph.D degrees from Chalmers Institute of Technology, Gothenburg, Sweden, in 1967 and 1970, respectively. From 1970 to 1979 he was affiliated with the Central Research and Development Laboratories of ASEA AB, Sweden, where he initiated and led the development of large computer based real–time computer systems for electric utilities, intelligent controllers, and mathematical software. He wrote one of the first sparse matrix software packages in the industry in 1970 – 1972, and developed parallel algorithms for scientific applications in the mid 1970s. From 1979 to 1983 he was a Senior Research Associate in computer science at the California Institute of Technology, where he taught VLSI design and started a course in Scientific Computing on Parallel Architectures. He joined the faculty of Yale University in 1983, where he introduced courses on parallel algorithms and architectures, and continued his research on architectures, algorithms and software for high performance parallel computers for the computational sciences. Since 1986 Dr. Johnsson is Director of Computational Sciences at Thinking Machines Corporation, where he initiated the development of a refined instruction set for the Connection Machine models CM–2 and CM–200, and is leading the development of efficient communication routines, and the development of the Connection Machine Scientific Software Library, CMSSL. Since 1990, Dr. Johnsson is Gordon McKay Professor of the Practice of Computer Science at Harvard University, where he again introduced education in scientific computation on parallel scalable architectures.

Dr. Johnsson is an editor of the *Journal of Parallel and Distributed Computing*, the *Journal of High Speed Computing*, the *Journal of Concurrency: Practice and Experience*, the *Journal of Numerical Linear Algebra with Applications*, the *International Journal of Supercomputer Applications*, and the *Journal of Scientific Programming.* He is the author or co–author of over 90 articles and conference papers, and numerous technical reports. He is the co–recipient of the 1986 Outstanding Paper Award at the International Conference on Par-

Table 11: Estimation of the number of cycles for matrix-vector multiplication (part 1).

loop ordering	Load/Store cycles	DRAM page faults
*PQPQM: A	$P\times Q\times M$	$((\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{A_0\times P \bmod \alpha}{\mathcal{P}}\rfloor)Q+(\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor\lfloor\frac{Q}{\beta}\rfloor+\lfloor\frac{A_1\times Q \bmod \beta}{\mathcal{P}}\rfloor)\lceil\frac{P}{\alpha}\rceil+\lceil\frac{P}{\alpha}\rceil\lceil\frac{Q}{\beta}\rceil)\times M$
B	$Q\times M$	$((1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor)\lfloor\frac{Q}{\beta}\rfloor+\lceil\frac{B_0\times Q \bmod \beta}{\mathcal{P}}\rceil)\times M$
C	$2\times P\times M\times\lceil\frac{Q}{\beta}\rceil$	$2(\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{C_0\times P \bmod \beta}{\mathcal{P}}\rfloor+\lceil\frac{P}{\alpha}\rceil)\lceil\frac{Q}{\beta}\rceil\times M$
$\{\alpha,\beta\}$	{16,32}	
*PQQPM: A	$P\times Q\times M$	$((\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{A_0\times P \bmod \alpha}{\mathcal{P}}\rfloor)Q+(\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor\lfloor\frac{Q}{\beta}\rfloor+\lfloor\frac{A_1\times Q \bmod \beta}{\mathcal{P}}\rfloor)\lceil\frac{P}{\alpha}\rceil+\lceil\frac{P}{\alpha}\rceil\lceil\frac{Q}{\beta}\rceil)\times M$
B	$Q\times M\times\lceil\frac{P}{\alpha}\rceil$	$((1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor)\lfloor\frac{Q}{\beta}\rfloor+\lceil\frac{B_0\times Q \bmod \beta}{\mathcal{P}}\rceil)\lceil\frac{P}{\alpha}\rceil\times M$
C	$2\times P\times M$	$2(\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{C_0\times P \bmod \beta}{\mathcal{P}}\rfloor+\lceil\frac{P}{\alpha}\rceil)\times M$
$\{\alpha,\beta\}$	{32,16}	
QPPQM: A	$P\times Q\times M$	$((\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor\lfloor\frac{Q}{\beta}\rfloor+\lfloor\frac{A_1\times Q \bmod \beta}{\mathcal{P}}\rfloor)P+(\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{A_0\times P \bmod \alpha}{\mathcal{P}}\rfloor)\lceil\frac{Q}{\beta}\rceil+\lceil\frac{P}{\alpha}\rceil\lceil\frac{Q}{\beta}\rceil)\times M$
B	$Q\times M$	$((1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor)\lfloor\frac{Q}{\beta}\rfloor+\lceil\frac{B_0\times Q \bmod \beta}{\mathcal{P}}\rceil)\times M$
C	$2\times P\times M\times\lceil\frac{Q}{\beta}\rceil$	$2(\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{C_0\times P \bmod \beta}{\mathcal{P}}\rfloor+\lceil\frac{P}{\alpha}\rceil)\lceil\frac{Q}{\beta}\rceil\times M$
cleanup	$P\times Q\times M$	
$\{\alpha,\beta\}$	{4,8},{8,4}	
*QPQPM: A	$P\times Q\times M$	$((\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor\lfloor\frac{Q}{\beta}\rfloor+\lfloor\frac{A_1\times Q \bmod \beta}{\mathcal{P}}\rfloor)P+(\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{A_0\times P \bmod \alpha}{\mathcal{P}}\rfloor)\lceil\frac{Q}{\beta}\rceil+\lceil\frac{P}{\alpha}\rceil\lceil\frac{Q}{\beta}\rceil)\times M$
B	$Q\times M\times\lceil\frac{P}{\alpha}\rceil$	$((1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor)\lfloor\frac{Q}{\beta}\rfloor+\lceil\frac{B_0\times Q \bmod \beta}{\mathcal{P}}\rceil)\lceil\frac{P}{\alpha}\rceil\times M$
C	$2\times P\times M$	$2(\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor\lfloor\frac{P}{\alpha}\rfloor+\lfloor\frac{C_0\times P \bmod \beta}{\mathcal{P}}\rfloor+\lceil\frac{P}{\alpha}\rceil)\times M$
cleanup	$P\times b\times M$	
$\{\alpha,\beta\}$	{4,8},{8,4}	
MQPPQM: A	$P\times Q\times M$	$(1+\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor a+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor ab)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q\times M$	$(1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor b)\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2\times P\times M\times\lceil\frac{Q}{\beta}\rceil$	$2(1+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor a)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{2,4,8},{4,2,8}	
*MQPQPM: A	$P\times Q\times M$	$(1+\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor a+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor ab)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q\times M\times\lceil\frac{P}{\alpha}\rceil$	$(1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor b)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2\times P\times M$	$2(1+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor a)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{2,4,8},{4,2,8}	

Table 12: Estimation of the number of cycles for matrix-vector multiplication (part 2).

loop ordering	Load/Store cycles	DRAM page faults
MPQPQM: A	$P{\times}Q{\times}M$	$(1+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor b\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor ab)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+$
		$o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q{\times}M$	$(1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor b)\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2{\times}P{\times}M{\times}\lceil\frac{Q}{\beta}\rceil$	$2(1+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor a)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{2,4,8},{4,2,8}	
*MPQQPM: A	$P{\times}Q{\times}M$	$(1+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor b+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor ab)+$
		$\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q{\times}M{\times}\lceil\frac{P}{\alpha}\rceil$	$(1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor b)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2{\times}P{\times}M$	$2(1+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor a)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{2,4,8},{4,2,8}	
PMQPQM: A	$P{\times}Q{\times}M$	$(1+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor b+$
		$\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor bk)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q{\times}M$	$(1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor b)\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2{\times}P{\times}M{\times}\lceil\frac{Q}{\beta}\rceil$	$2(1+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor k)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{8,2,4},{8,4,2}	
PMQQPM: A	$P{\times}Q{\times}M$	$(1+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor b+$
		$\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor bk)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q{\times}M{\times}\lceil\frac{P}{\alpha}\rceil$	$(1+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor b)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2{\times}P{\times}M$	$2(1+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor k)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{8,2,4},{8,4,2}	
QMPPQM: A	$P{\times}Q{\times}M$	$(1+\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor a+$
		$\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor ak)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q{\times}M$	$(1+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor k)\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2{\times}P{\times}M{\times}\lceil\frac{Q}{\beta}\rceil$	$2(1+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor a)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
cleanup	$P{\times}Q{\times}M$	
$\{\alpha,\beta,\gamma\}$	{8,2,4},{8,4,2}	
QMPQPM: A	$P{\times}Q{\times}M$	$(1+\lfloor\frac{A_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{A_2\times k}{\mathcal{P}}\rfloor a+\lfloor\frac{A_1\times\beta}{\mathcal{P}}\rfloor ak)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+$
		$o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
B	$Q{\times}M{\times}\lceil\frac{P}{\alpha}\rceil$	$(1+\lfloor\frac{B_2\times k}{\mathcal{P}}\rfloor+\lfloor\frac{B_0\times\beta}{\mathcal{P}}\rfloor k)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{Q}{\beta}\rfloor\lfloor\frac{M}{k}\rfloor)$
C	$2{\times}P{\times}M$	$2(1+\lfloor\frac{C_0\times\alpha}{\mathcal{P}}\rfloor+\lfloor\frac{C_2\times k}{\mathcal{P}}\rfloor a)\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor+o(\lfloor\frac{P}{\alpha}\rfloor\lfloor\frac{M}{k}\rfloor)$
$\{\alpha,\beta,\gamma\}$	{8,2,4},{8,4,2}	
cleanup	$P{\times}b{\times}M$	

allel Processing, and a recipient of the John Ericsson Medal. Dr. Johnsson has served as a Board Member of the Computing Research Association and serves on the USRA Science Council for CESDIS. Dr. Johnsson has served on the program and organizing committees for several conferences on parallel computing. This work was conducted when author was with Thinking Machines Corporation and Harvard University, Cambridge, Massachusetts.

Yu Hu

Yu Hu received the B.S. degree from the University of Science and Technology of China in 1989, and the M.S. degree in Computer Science from Yale University in 1992. He is now a doctoral candidate in Computer Science at Harvard University. His research interests are in the fields of languages, compilers, and architectures for parallel computing, with a special interest in scientific applications.

Mr. Hu is a member of the ACM and IEEE.

References

Choi J., Dongarra J., Pozo R. and Walker D, 1992. ScaLAPACK: A Scalable Linear Algebra for Distributed , Memory Concurrent Computers, CS-92-181, University of Tennessee.

Comerford R. and Watson G. F, 1992. Memory catches up, *IEEE Spectrum*, 29(10):34–35.

Dongarra J. J., Bunch J., Moler C. and Stewart G.,1988a. An extended set of basic linear algebra subprograms, *ACM TOMS*, 14(1):1–17.

Dongarra J. J., Bunch J., Moler C. and Stewart G., 1988b. An extended set of basic linear algebra subprograms: Model implementation and test programs, *ACM TOMS*, 14(1):18–32.

Dongarra J. J., Du Croz J., Hammarling S. and Duff I., 1990a. A set of level 3 basic linear algebra subprograms: Model implementation and test programs, *ACM TOMS*, 16(1):18–28.

Dongarra J. J., Du Croz J., Hammarling S. and Duff I., 1990b. A set of level 3 basic linear algebra subprograms, *ACM TOMS*, 16(1):1–17.

Farmwald M. and Morning D., 1992. A fast path to one memory, *IEEE Spectrum*, 29(10):50–51.

High Performance Fortran Forum, 1993. High Performance Fortran; Language Specification, Version 1.0, *Scientific Programming*, 2(1 - 2):1–170.

Hingham N. J., 1992. Stability of a method for multiplying complex matrices with three real matrix multiplications, *SIAM Journal on Matrix Analysis and Applications*, 13(3):681–687.

Hockney R. W. and Jesshope C. R., 1981. *Parallel Computers*, Adam Hilger.

Johnsson S. L. and Mathur K. K., 1992., Distributed BLAS, Thinking Machines Corp., In preparation.

Johnsson S. L. and Ortiz L. F., 1992. Local Basic Linear Algebra Subroutines (LBLAS) for Distributed Memory Architectures and Languages with an Array Syntax, *The International Journal of Supercomputer Applications*, 6(4):322–350.

Lawson C. L., Hanson R. J., Kincaid D. R., Krogh F. T., 1979a. Algorithm 539: Basic Linear Algebra Subprograms for Fortran Usage, *ACM TOMS*, 5(3):324–325.

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

Mathur K. K. and Johnsson S. L., 1994. Multiplication of Matrices of Arbitrary Shape on a Data Parallel Computer, *Parallel Computing*, 20(7):919–951.

Metcalf M. and Reid J, 1991. *Fortran 90 Explained*, Oxford Scientific Publications.

Olsson P. and Johnsson S. L. 1990. A Data-parallel Implementation of Explicit Methods for the Three-dimensional Compressible Navier-Stokes Equations, *Parallel Computing*, 14(1):1–30.

Thinking Machines Corporation, 1991. *CM–5 Technical Summary.*

Thinking Machines Corporation, 1993. *CMSSL for CM Fortran, Version 3.1*

Thinking Machines Corporation, 1994. *The CM Run–Time System (CMRTS).*