Implementing BLAS Level 3 on the CAP-II*

Peter E. Strazdins and Richard P. Brent†

Department of Computer Science
and
Computer Sciences Laboratory
Australian National University

Abstract

The Basic Linear Algebra Subprogram (BLAS) library is widely used in many supercomputing applications, and is used to implement more extensive linear algebra subroutine libraries, such as LINPACK and LAPACK. The use of BLAS aids in the clarity, portability and maintenance of mathematical software. BLAS level 1 routines involve vector-vector operations, level 2 routines involve matrix-vector operations, and level 3 routines involve matrix-matrix operations. To take advantage of the high degree of parallelism of architectures such as the CAP-II, BLAS level 3 routines are desirable. These routines are not I/O bound; for $n \times n$ matrices, the order of arithmetic operations is $O(n^3)$ whereas the order of I/O operations is only $O(n^2)$.

We are concerned with implementing BLAS level 3 for real matrices on the CAP-II, with emphasis on obtaining the highest possible performance, without sacrificing numerical stability. While the CAP-II has many features that make it very well-suited for this purpose, there are also many new challenges in implementing BLAS level-3 on a distributed memory parallel computer (these are currently being considered also by the authors of BLAS-3, who designed it primarily for cache and shared memory architectures).

One such challenge is its external interface: BLAS-3 subroutines can be called by the host program, with the CAP array used like a (very powerful) floating point unit. Alternatively, (CAP cell equivalents of) BLAS-3 subroutines may be called by the cell programs; this approach is more efficient but deviates from the BLAS standards. These issues will be discussed.

We also discuss the high-level design of basic parallel matrix multiplication, transposition and triangular matrix inversion algorithms to be used by the BLAS-3 subroutines. With the efficient row/column broadcast available on the CAP-II using wormhole routing, "semi-systolic" algorithms, with a low startup time, appear to be superior to other algorithms. It is hoped that the input from the workshop can help to finalize details of the high-level design.

On the lower levels of design, optimization of cell program codes (e.g. optimizing inner product or gaxpy loops, use of the SPARC cache) must also be considered. Also relevant is the degree of optimization possible from the available CAP-II cell program compilers.

*Appeared in *Proc. First Fujitsu-ANU CAP Workshop* (edited by R. P. Brent and M. Ishii), Fujitsu Research Laboratories, Kawasaki, Japan, November 1990.

†E-mail addresses: {peter,rpb}@cs.anu.edu.au

rpb121 typeset using LaTeX

1 Introduction

Libraries of linear algebra routines such as LINPACK [6] and LAPACK [1, 4] have been implemented using certain basic linear algebra subprograms (BLAS) as primitives. The motivation for this is a combination of portability and efficiency – the high-level routines are written in a machine-independent manner in a widely-available language (usually Fortran 77), but the BLAS may be coded in a machine-dependent manner in a high or low-level language. When porting the linear algebra libraries to a new machine, it is easy to get them running by using a portable (but possibly inefficient) implementation of the BLAS. Higher efficiency can then be obtained by recoding the BLAS to take better advantage of the machine architecture.

1.1 Different Levels of BLAS

Three distinct levels of BLAS have been proposed:

Level 1 BLAS are the original BLAS [6, 13, 14] and implement elementary vector operations, e.g.

$$y \leftarrow \alpha x + y$$

and

$$s \leftarrow x^T y$$

(where x and y are vectors of the same dimension n, and α, s are scalars). Such operations involve $O(n)$ arithmetic operations on $O(n)$ data items.

Level 2 BLAS [9, 10] implement matrix-vector operations, e.g.

$$y \leftarrow \alpha Ax + \beta y$$

(where A is a matrix, x and y are vectors of compatible dimensions – for simplicity we assume here that A is n by n). Such operations involve $O(n^2)$ arithmetic operations on $O(n^2)$ data items. Use of level 2 BLAS can give greater efficiency on vector processors than is possible with level 1 BLAS, because of the reduction in procedure calling overheads, vector register loads/stores, etc. Implementations of level 2 BLAS on MIMD machines such as the CAP-II could be inefficient because a large proportion of the execution time would be spent transferring data between cell processors or between the host and cell processors.

Level 3 BLAS [7, 8] implement matrix-matrix operations, e.g.

$$C \leftarrow \alpha AB + \beta C$$

(where A, B and C are compatible matrices – for simplicity we assume here that they are n by n). Such operations involve $O(n^3)$ arithmetic operations on $O(n^2)$ data items. (We ignore the possibility of reducing the number of operations by the use of "fast" matrix multiplication algorithms: this may be of practical use if n is large [3, 5, 11, 15].) Note that there is a higher ratio of arithmetic operations to data than for the level 2 BLAS. Use of level 3 BLAS is attractive on parallel machines such as the CAP-II because the cost of a data transfer may be amortised over the cost of $O(n)$ arithmetic operations.

1.2 The Level 3 BLAS

The Level 3 BLAS (or *BLAS3* for short) are fully described in [7, 8]. We restrict our attention to operations on real matrices, since complex matrices introduce some inessential complications. The BLAS3 perform matrix-matrix multiply-and-add operations of the form

$$C \leftarrow \alpha \tilde{A}\tilde{B} + \beta C$$

where $\tilde{A}$ can be either A or A^T (and similarly for $\tilde{B}$),

$$C \leftarrow \alpha A A^T + \beta C$$

where C is symmetric (and other forms of symmetric update),

$$B \leftarrow \alpha \tilde{T} B$$

where T is triangular and $\tilde{T}$ can be T, T^T, T^{-1} or T^{-T}, and

$$B \leftarrow \alpha B \tilde{T}$$

Matrices may be general rectangular, symmetric or triangular but there is no special form of "packed" storage for symmetric or triangular matrices.

Within each of the six BLAS3 routines, there are either 4 or 8 different matrix 'orientations', each requiring a different variant of the basic algorithm used. Combining this factor with the requirement that the BLAS3 routines, to be widely used by LAPACK, operate on matrix 'sub-blocks' rather than on whole matrices themselves (see Figure 1), the difficulties in implementation are considerable. Thus, some authors have suggested that BLAS3 routines on a distributed memory computer should be implemented in terms of a matrix-matrix multiply routine and the Level 2 BLAS [2, p10]. Our approach is not so extreme and it involves the BLAS3 routines calling a small library of more primitive matrix-matrix routines, so that the potentially large code size required by the BLAS3 routines can be kept manageable.

2 The External Interface

On a single processor the interface to the BLAS3 is via straightforward procedure calls. On a machine such as the CAP-II there are several possibilities because the host and cell processors have independent programs and memories. We have considered three possibilities –

1. Host only. BLAS3 are called only from host programs, and all matrices are stored on the host. The implementation may scatter data to cells and gather results in the host, but this should be transparent to the user. This approach is convenient for a user writing host programs, but potentially inefficient because of data transfer overheads.

2. Host and cell. BLAS3 are called from the host, but operate on matrices stored in the cell processors. As for case 1, the host must broadcast a request to the cells to perform the appropriate local computations by calling the appropriate cell subroutines.

3. Cell only. BLAS3 are called from cell programs only, and operate on matrices stored in the cell processors. As for case 2, a standard storage scheme has to be assumed, and routines which move arrays between the host and cells are necessary. It is assumed that the cells execute identical programs. This scheme has the potential to be the most efficient.

At this date, the authors of BLAS are still considering which external interface should be used for distributed memory computers. While it seems likely that interface 2 will be the most acceptable, we are pursuing interface 3 (which can be made into interface 2 if needed). For implementing BLAS3 on a large CAP (eg. $8 \times 8, 16 \times 16$ or 32×32 CAP), there are two reasons to prefer interfaces 2 and 3:

- With the assumption that 8M of data (matrix) memory be available in each CAP cell, the total memory on the CAP array will easily exceed that on the CAP host.

- On a 16×16 CAP, 1GFLOP is a realistic target for BLAS3 performance. Assuming a 50MBs^{-1} host to CAP array I/O bandwidth, and BLAS3 routines operating on $n \times n$ matrix sub-blocks, then $n \geq 2^8$ before computation time exceeds host to array I/O transfer time.

Furthermore, while a 32×32 CAP has ≈ 4 times the peak computational performance of the CRAY Y-MP/8, the CRAY Y-MP/8 has an exceptionally high memory to CPU I/O bandwidth of 16000MBs^{-1}. In order for the CAP-II to exceed the CRAY Y-MP/8 in overall performance, interfaces 2 or 3 must be chosen.

3 High Level Design Choices

In this section we consider some choices which have to be made regarding storage of matrices in cells, the multiplication and the transposition of matrices, etc.

3.1 Distribution of Matrices

Matrices could be distributed over cells of the CAP-II by rows, by columns, by contiguous blocks, or by the *cut and pile* strategy, in which matrix element $a_{i,j}$ is stored in cell $(i \bmod s, j \bmod t)$, assuming that there are $s \times t$ cells in the CAP array (see Figure 1). The cut and pile strategy is similar to the "dot mode" of image storage used in [12] and has similar advantages. It gives a good load distribution when performing operations on triangular or symmetric matrices (since each CAP cell would have a triangular sub-block). Thus, for efficiency we prefer the cut and pile strategy, even though it is inconvenient for matrix representation on the host (so conversion may be required before/after scatter/gather operations).

c_{00}	c_{02}	c_{04}	c_{06}	c_{01}	c_{03}	c_{05}	c_{07}
c_{20}	c_{22}	c_{24}	c_{26}	c_{21}	c_{23}	c_{25}	c_{27}
c_{40}	c_{42}	c_{44}	c_{46}	c_{41}	c_{43}	c_{45}	c_{47}
c_{60}	c_{62}	c_{64}	c_{66}	c_{61}	c_{63}	c_{65}	c_{67}
c_{10}	c_{12}	c_{14}	c_{16}	c_{11}	c_{13}	c_{15}	c_{17}
c_{30}	c_{32}	c_{34}	c_{36}	c_{31}	c_{33}	c_{35}	c_{37}
c_{50}	c_{52}	c_{54}	c_{56}	c_{51}	c_{53}	c_{55}	c_{57}
c_{70}	c_{72}	c_{74}	c_{76}	c_{71}	c_{73}	c_{75}	c_{77}

Figure 1: Distribution of an 8×8 matrix C on a 2×2 CAP; C' is the 4×4 matrix sub-block of C whose upper-left corner is at position $(2, 4)$

3.2 Parallelism within or outside of BLAS3 routines

For implementing BLAS3 on the CAP, parallelism can be used in two ways:

- using all CAP cells in parallel to execute a single BLAS-3 routine call.
- executing independent BLAS-3 routine calls over different groups of CAP-II cells.

While both ways are important, the second is difficult on the CAP since the basic matrix multiplication and transposition algorithms (see below) use the whole of the torus network. The choice of the cut-and-pile matrix distribution makes this way even more difficult.

Thus, we design the BLAS3 for the CAP so that only parallelism within the BLAS3 routines is used.

3.3 Matrix Transposition

Most of the BLAS3 routines require the ability to form matrix products using operands which may be transposed. It is not obvious whether or not matrix operands should be explicitly transposed: by taking advantage of the wormhole routing and torus topology of the CAP-II, it should be possible to implement a fast matrix transpose. On the other hand, the overhead involved in avoiding transposition is probably very low if only one operand must be transposed, at least if $s = t$ and a "semi-systolic" matrix multiplication scheme is used (see below).

It seems better, however, to use explicit matrix transposition to form $C \leftarrow \alpha A^T B^T + \beta C$, using the fact that $C' = A^T B^T = (BA)^T$. Here, the cell matrix sub-blocks of $C'' = BA$ from each CAP cell must be passed through the main diagonal cell (on the same row) before reaching its destination. Since the CAP Routing Controller only allows one message to pass through a cell when the message is changing its direction, the main diagonal cells form the bottleneck for the transpose. Thus, the simplest matrix transposition algorithm is probably the most efficient:

r_asend(getncely(), getncelx(), tid, ANY_TYPE, $C'', m * n$);
C' = r_arecv(getncely(), getncelx(), tid, ANY_TYPE);

For the symmetric matrix multiplication, and for multiplication by a transposed inverse of a triangular matrix, 'half-transpose' operations (eg. copying a lower triangular matrix into its upper triangular half) are required; in this case, greater efficiency can be obtained by routing half of the messages in two stages (first vertically, then horizontally) to ensure that the same number of messages pass through each main diagonal cell.

3.4 Matrix Multiplication

As mentioned above, BLAS3 routines operate on sub-blocks of global matrices (eg. C' in Figure 1), rather than whole matrices as such. The part of C' in each cell is generally not contiguous, and for the requirements of message-passing, must be *compressed* into a contiguous form first. Thus, for any matrix multiplication operation of the form $C \leftarrow \alpha\tilde{A}\tilde{B} + \beta C$, the global high-level algorithm is as follows:

compress A, B sub-blocks into A', B';
$C'' \leftarrow \tilde{A}'\tilde{B}'$;
$C \leftarrow \alpha$(de-compressed version of C'') $+ \beta C$

Thus, storage needs to be allocated for the 'temporary' sub-blocks A', B' and C''.

Furthermore, the optimal matrix multiplication algorithm depends on the sub-block size. For large sub-block sizes, the 'inner product' algorithm is optimal; for (very) small block sizes (ie. $<$ getncelx()), an 'outer product' algorithm, like that used for matrix inversion (next section), should be considered.

Consider the formation of the matrix $C = AB$ in terms of its cell sub-blocks C_{ij}:

$$C_{ij} = \Sigma_{k=0}^{N-1} A_{ik} B_{kj}$$

where N = getncelx(). Here, the (i, j)th cell sub-block corresponds to that allocated on CAP cell (i, j)[1].

The full systolic method consists of simulating a skewed input of A from the east, and that of B from the north, and accumulating C_{ij} in cell (i, j) as each A_{ik} and B_{kj} passes through. This method involves only shifting of the cell sub-blocks, but has a considerable startup overhead (in the simulated matrix input).

The semi-systolic method also consists of accumulating C_{ij} in cell (i, j) but using the products $A_{i,j+k} B_{j+k,j}$ (addition is cyclic modulo the CAP array size). This involves, on the kth step, shifting the (rows of the) A cell sub-blocks one unit west, and a column broadcast of the kth diagonal of B (the 0th diagonal is the main diagonal, the first is the one below concatenated with the upper-right corner element, etc). The high-level cellprogram algorithm is as follows:

[1]The matrix multiplication and transposition algorithms are the same for the cut-and-pile and the contiguous matrix distribution strategies. It will be easier, for this section, to assume that the contiguous strategy is used.

B_{00}	B_{11}	B_{22}
A_{00}	A_{01}	A_{02}
A_{10}	A_{11}	A_{12}
A_{20}	A_{21}	A_{22}

(a) k=0

B_{10}	B_{21}	B_{02}
A_{01}	A_{02}	A_{00}
A_{11}	A_{12}	A_{10}
A_{21}	A_{22}	A_{20}

(b) k=1

B_{20}	B_{01}	B_{12}
A_{02}	A_{00}	A_{01}
A_{12}	A_{10}	A_{11}
A_{22}	A_{20}	A_{21}

(c) k=2

Figure 2: Formation of $C = AB$ on a 3×3 CAP (note: C_{ij} is in cell (i, j))

```
for (k=0; k < N-1; k++)
    y-broadcast B cell sub-block from kth diagonal cells;
    send A cell sub-block to western neighbour cell;
    perform local cell sub-block multiplication;
    recv new A cell sub-block from eastern neighbour cell;
y-broadcast B cell sub-block from (N-1)th diagonal cells;
perform local cell sub-block multiplication;
```

The disadvantages of this method is the (small) extra overhead in using broadcasts (rather than send to neighbours), and also that the broadcasts cannot be overlapped with the local cell sub-block multiplication. Figure 2 shows the algorithm for a 3×3 CAP.

There are full systolic methods for computing $C = AB^T$ ($C = A^T B$) without requiring any explicit transpose; here, the result matrix and one of the operand matrices are rotated, and the overhead is a small amount of extra start-up.

The semi-systolic method for computing $C = AB^T$ ($C = A^T B$) is performed by adding the product $A_{i,j-k}(B_{j,j-k})^T$ $((A_{i+k,i})^T B_{i+k,j})$ to C_{ij}. Here, only local transposition of cell matrix sub-blocks is required and the C matrix is cyclically shifted at each step one unit west (south), and the kth diagonal of B ($-k$th diagonal of A) being column (row) broadcasted. The only extra overhead involved here is that it is more difficult to overlap the communication of the C matrix with the local cell sub-block multiplication.

However, for the BLAS-3 symmetric update routines, since the result matrix is symmetric and need only be represented by its upper or lower triangular part, avoiding explicit transpose has the advantage that inter-cell communication is reduced by 25% to 37.5%.

3.5 Triangular matrix inversion

Consider multiplying a rectangular matrix B by the inverse of a triangular matrix, A. Firstly, we require the possible scaling of A (and hence B) so that A's diagonal is unit (in BLAS3, it is up to the user to ensure that this is possible, ie. that A is non-singular).

Let n be the number of rows or columns in A. If A is upper triangular, a column of zeroes can be introduced in column J, $J = n-1, ..., 1$ in parallel by the row broadcasting of the Jth column of A, $A_{.J}$ (and using the Jth row of A, $A_{J.}$; this need not be column broadcast since here $A_{J,k} = 1.0$ if $k = J$ and is 0.0 otherwise); then, the 'outer product' update $A \leftarrow A - A_{.J}A_{J.}$ is performed. Each cell communicates $O(k)$ data per $O(k^2)$ arithmetic operations, where $k = n/\text{getncelx}()$ is the cell sub-block size, so the algorithm is efficient.

The $A_{.J}$ correspond to a pre-multiplication by a 'parallel' elementary matrix (row update) operation matrix $E_{.J}$, so that $A^{-1} = E_{1.} \ldots E_{(n-1).}$. Thus, using this to form $B \leftarrow A^{-1}B$, we can perform the corresponding (parallel) row updates on B:

$$B \leftarrow B - A_{.J}B_{J.}, \quad \text{for } J = n-1, ..., 1$$

Similarly, we can introduce a row of zeroes in row $I, I = 0, ..., N-2$ by the operation $A \leftarrow A - A_{.I}A_{I.}$; these operations correspond to a post-multiplication by a 'parallel' elementary matrix (column update) operation matrix $E_{I.}$, so that $A^{-1} = E_{0.} \dots E_{(n-2).}$. Thus, using this to form $B \leftarrow BA^{-1}$, we can perform the corresponding (parallel) column update on B:

$$B \leftarrow B - B_{.I}A_{I.}, \quad \text{for } I = 0, ...n-2$$

In this way, we can use the efficient row/column broadcast mechanisms in the CAP to perform the 'outer product' update required, without the explicit formation of A^{-1}. Outer product updates may also be efficient for the multiplication of small matrix sub-blocks also.

4 Implementation issues

In this section, we discuss our ideas on the implementation of the BLAS3 in the CAP-II.

4.1 Source Language

Currently, we are using C as the source language; the reasons for this are that it has superior handling of pointers, which may enhance code optimization. Note that the CAP-II cells represent local matrix sub-blocks as one-dimesional arrays.

Since the SPARC provides, via 'register windows', very efficient subroutine call support, the BLAS3 routines are organized so that common code is expressed as subroutines. This is to reduce overall code size, a reasonably important consideration.

4.1.1 Workspace

Because of the communication protocols on the CAP-II, temporary storage for matrix sub-blocks is required for their communication. This is at odds with the design of the authors of BLAS3 [1], who avoid (and see as undesirable) any large temporary storage.

Currently, all BLAS3 routines share static temporary matrix sub-block workspaces declared externally (by the cellprogram); this is sufficient to store three sub-blocks of the largest required size.

4.2 Local matrix product formation

Since SPARC chips have pipelined instruction sets and (a small number of) parallel floating point coprocessors, a 'gaxpy' form (ie. i, k, j is the order of nesting of loops used to form $c_{ij} = \Sigma_{k=0}^{n-1} a_{ik} b_{kj}$).

Loop 'unrolling' may be an optimization which can speed up this process. Also, to minimize cache misses (SPARC cache is 128K), the product formation can be broken down into the formation of 32K sub-blocks; here, the 'inner product' (ie. i, j, k) loop order is used. Both of these optimizations are regarded as advanced and will not be implemented until later.

5 Conclusions

Many of the CAP's architectural features, such as wormhole routing (with efficient row/column broadcasts), large cell memories and, most of all, the potential for massive parallelism make it extremely promising for implementing BLAS3 for supercomputing applications. This is especially the case for large CAP and matrix sizes, for which our design is intended. The CAP-II can also exploit most of the BLAS3 features that are motivated by efficiency; the main exception being the CAP's need for temporary matrix workspace (an inevitable result of asynchronous communication on a distributed memory machine). While an optimized implementation of the BLAS3 is not easy, we feel that it would greatly enhance the CAP-II's usability in supercomputing applications.

References

[1] C. Bischof, J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling and D. C. Sorensen, *Provisional Contents,* LAPACK Working Note #5, Report ANL-88-38, Mathematics and Computer Science Division, Argonne National Laboratory, September 1988.

[2] C. Bischof, *Fundamental Linear Algebra Computations on High-Performance Computers,* Preprint MCS-P150-0490, Mathematics and Computer Science Division, Argonne National Laboratory, August 1990.

[3] R. P. Brent, *Algorithms for Matrix Multiplication,* Report STAN-CS-70-157, Department of Computer Science, Stanford University, March 1970.

[4] J. W. Demmel, J. J. Dongarra, J. J. Du Croz, A. Greenbaum, S. J. Hammarling and D. C. Sorensen, *Prospectus for the Development of a Linear Algebra Library for High Performance Computers,* Report ANL-MCS-TM-97, Mathematics and Computer Science Division, Argonne National Laboratory, September 1987.

[5] J. W. Demmel and N. J. Higham, *Stability of Block Algorithms with Fast Level 3 BLAS*, preprint, July 1990.

[6] J. J. Dongarra, J. Bunch, C. Moler and G. W. Stewart, *LINPACK Users' Guide,* SIAM, Philadelphia, Pa., 1979.

[7] J. J. Dongarra, J. J. Du Croz, S. J. Hammarling and I. S. Duff, "A set of level 3 basic linear algebra subprograms", *ACM Transactions on Mathematical Software* 16, (1990), 1–17.

[8] J. J. Dongarra, J. J. Du Croz, S. J. Hammarling and I. S. Duff, "Algorithm 679: A set of level 3 basic linear algebra subprograms: model implementation and test programs", *ACM Transactions on Mathematical Software* 16 (1990), 18–28.

[9] J. J. Dongarra, J. J. Du Croz, S. J. Hammarling and R. Hanson, "An extended set of Fortran basic linear algebra subprograms", *ACM Transactions on Mathematical Software* 14 (1988), 1–17.

[10] J. J. Dongarra, J. J. Du Croz, S. J. Hammarling and R. Hanson, "An extended set of Fortran basic linear algebra subprograms: model implementation and test programs", *ACM Transactions on Mathematical Software* 14 (1988), 18–23.

[11] N. J. Higham, *Exploiting Fast Multiplication within the Level 3 BLAS,* Technical Report 89-984, Department of Computer Science, Cornell University, 1989 (to appear in *ACM Transactions on Mathematical Software*).

[12] M. Ishii, G. Goto and Y. Hatano, "Cellular array processor CAP and its application to computer graphics", *Fujitsu Scientific and Technical Journal* 23 (1987), 379–390.

[13] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, "Basic linear algebra subprograms for Fortran usage", *ACM Transactions on Mathematical Software* 5 (1979), 308–323.

[14] C. Lawson, R. Hanson, D. Kincaid and F. Krogh, "Algorithm 539: Basic linear algebra subprograms for Fortran usage", *ACM Transactions on Mathematical Software* 5 (1979), 324–325.

[15] V. Strassen, "Gaussian elimination is not optimal", *Numerische Mathematik* 13 (1969), 354-356.