The Implementation of BLAS level 3 on the AP1000: Preliminary Report*

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. To take advantage of the high degree of parallelism of architectures such as the Fujitsu AP1000, BLAS level 3 routines (matrix-matrix operations) are proposed.

This project is concerned with implementing BLAS level 3 (BLAS-3) for single precision matrices on the AP1000, with emphasis on obtaining the highest possible performance, without significantly sacrificing numerical stability. This paper discusses the techniques used to achieve this goal, together with the underlying issues.

The most important techniques were the use of software pipelining and loop unrolling for writing optimized assembler inner loops for matrix inner and outer products, which were able to operate at more than 90% and 70%, respectively, of the AP1000's theoretical peak performance.

The efficiency of cell communication using wormhole routing on the AP1000, especially the row/column broadcast, enabled a sustained performance of 80 to 90% of the theoretical peak for all the BLAS-3 routines. It also meant that many variations (using different communication schemes) for matrix multiplication have more or less equivalent performance. However, for future versions of the AP1000, optimizing communication must still be considered.

Techniques for improving the performance for large matrices (partitioning, to improve cache utilization) and for small matrices (minimizing communication) are employed. The latter have been developed for general rectangular AP1000 configurations.

1 Introduction

Libraries of linear algebra routines such as LINPACK [6] and LAPACK [1, 5] 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.

*Appeared in *Proceedings of the Second Fujitsu-ANU CAP Workshop* (edited by R. P. Brent), Department of Computer Science, ANU, November 1991, H1–H17.

†E-mail addresses: {peter,rpb}@cs.anu.edu.au rpb131 typeset using LaTeX

The Fujitsu AP1000 [12, 13] is a scalable array processor using wormhole routing and high-performance SPARC-based cells. It is a distributed memory architecture with attributes that should allow an efficient implementation of the BLAS level 3 (BLAS-3) procedures.

This project has the following purposes. Firstly, to evaluate the suitability of the AP1000 as a numeric supercomputer. This needs some explanation: the BLAS-3 are increasingly regarded as a "benchmark" for the numeric performance of modern supercomputer architectures. This is because a well designed and balanced architecture must be able to achieve high performance on the BLAS-3 if it expects to be able achieve high performance on more complex numerical algorithms.

The second purpose is discover and compile the techniques used to obtain this performance on the AP1000, which can assist the design of more complex numerical algorithms on the AP1000. The third is to evaluate the suitability of the BLAS-3 itself on distributed memory architectures, an issue only beginning to be addressed recently, as BLAS-3 was designed for shared memory or cache-based processors. The final purpose is to be able to provide users with a workable numerical library on the AP1000 which can reduce program development time and still yield good performance.

1.1 The BLAS Level 3

BLAS Level 3 [7, 8] implement matrix-matrix operations, which, for n by n, matrices, involve $O(n^3)$ arithmetic operations on $O(n^2)$ data items. This yields is a higher ratio of arithmetic operations to data than for the BLAS level 2 (BLAS-2) [9, 10], although degenerate cases of the BLAS-3 routines yield all BLAS-2 routines. Use of BLAS-3 is attractive on parallel machines such as the AP1000 because the cost of a data transfer may be amortised over the cost of $O(n)$ arithmetic operations.

We restrict our attention to operations on single precision real matrices, since this is sufficient to illustrate the essential implementation issues. The BLAS-3 perform multiply-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 $\tilde{B}$), multiply-add operations for symmetric matrices, eg.:

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

where C is symmetric, and triangular matrix update operations for the form:

$$B \leftarrow \alpha \tilde{A} B, \;\; B \leftarrow \alpha B \tilde{A}$$

where A is triangular and $\tilde{A}$ can be A, A^T, A^{-1}. 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 BLAS-3 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 BLAS-3 routines operate on matrix "sub-blocks" rather than on whole matrices themselves, and should be efficient over a large range of matrix shapes and sizes, the difficulties in implementation are considerable. Our approach involves the BLAS-3 routines calling a small library of more primitive matrix-matrix routines, so that the potentially large code size required by the BLAS-3 routines can be kept manageable. It is the performance of these more primitive routines that will be reported in this paper.

2 High Level Design Choices

In this section, the high level choices for the design of BLAS-3 on the AP1000 are described.

2.1 The External Interface

In [16], the following options were considered for the external interface:

1. Host only. BLAS-3 are called only from host programs, and all matrices are stored on the host.

2. Host and cell. BLAS-3 are called from the host, but operate on matrices stored in the cell processors.

3. Cell only. BLAS-3 are called from cell programs only, and operate on matrices stored in the cell processors.

Option 3, being the simplest and most efficient, as well as offering the largest amount of memory, has been pursued. At this date, the authors of BLAS are "studying the options" for the interface of the BLAS-3 for distributed memory architectures [11]. Their intermediate goal is to design first the Basic Linear Algebra Communication subroutines (BLACS) [11], a set of subroutines for communicating matrices between cells. Although this issue is important from the user perspective, it is not so important from the point of view of implementation, so its resolution can be deferred until a later date.

2.2 Distribution of Matrices

To improve load balancing for operations on triangular matrices of sub-blocks of larger matrices, matrices are distributed over AP1000 cells by the *cut-and-pile* or *scattered* strategy, rather than storage by rows, by columns, or by contiguous blocks. In the scattered strategy, in which matrix element $a_{i,j}$ is stored in cell ($i \bmod N_y, j \bmod N_x$), assuming that there are $N_y \times N_x$ cells in the AP1000 array.

However, all the above strategies can be covered by an $\bar{m} \times \bar{n}$ "blocked panel-wrapped" strategy, which is sufficient for linear algebra applications in practice [11]. Here matrix element $a_{i,j}$ is stored in cell $((i/\bar{m}) \bmod N_y, (j/\bar{n}) \bmod N_x)$ of an $N_y \times N_x$ AP1000. In this case, the BLAS-3 routines themselves would have to know the appropriate storage parameters $(\bar{m}, \bar{n})$ for each matrix and manage the appropriate data movements. Since the complexities resulting from this extra flexibility are considerable, the implementation of a general matrix distribution strategy on the AP1000 is not considered in this paper.

On the cells, the matrix sub-blocks were stored in row-major order (C convention) rather than in column major order (the Fortran convention), although this decision does not greatly affect implementation issues. Associated with the row-major storage scheme for an $m \times n$ (cell sub-) matrix A is the *last dimension* of A, denoted ld_A, where $n <= \mathrm{ld}_A$. This enables A to be identified as a sub-matrix of a larger $m' \times \mathrm{ld}_A$ matrix A', where $m <= m'$.

2.3 Parallelism within or outside of BLAS-3 Routines

For implementing BLAS-3 on the AP1000, parallelism can be used in two ways:

- using all AP1000 cells in parallel to execute a single BLAS-3 routine call.
- executing independent BLAS-3 routine calls over different groups of AP1000 cells.

While some authors think the second way is important [2, 11], it introduces further complexities, especially with respect to matrix distribution, so only the first has been pursued in our implementation. It should be noticed that, for sufficiently large matrices, the second way is unlikely to offer a significant efficiency advantage.

K	16	32	48	64	80	96	112	128	160	192
speed	6.9	7.5	7.5	7.6	7.80	7.77	7.72	7.74	7.67	7.33

Table 1: Speed (MFLOPs) of `update4x4` loop-based $K \times K$ matrix multiply on a 1×1 AP1000

2.4 AP1000 Configuration

The AP1000 can be configured in a rectangular array of any shape. For a particular instance of a BLAS-3 operation, the optimal shape will generally be one close to the shape of its output matrix [14], assuming communication performance is the same on all AP1000 configurations. However, this assumption is not generally valid, and the "natural" configuration yields superior performance. Hence this option has been chosen for our implementation of the BLAS-3 for the current 8×8 AP1000 currently installed at ANU. The complexity of high-performance coding increases considerably for non-square configurations as is discussed in Section 3.3, but it is important that these be fully implemented in the near future.

3 Implementation of the BLAS-3

In this section, we look at low level design (or implementation) choices used to implement the BLAS-3 on the AP1000. Due to the high relative speed of communication to computation on the AP1000, efficiency depends on writing optimized assembler routines to perform matrix update inner loops, and then making appropriate "higher-level" choices regarding algorithm design and partitioning strategies.

3.1 High Performance Computation

Here, we discuss the main assembler routines ("loops") which form the core of the BLAS-3 computations, and then explain their implications.

3.1.1 The `update4x4` inner loop

For the matrix multiply-add operation $C \leftarrow C + AB$, where all matrices are $K \times K$, the most efficient inner loop for the SPARC cell architecture was found to be to update a 4×4 C sub-block $C_{i..i+3,j..j+3}$ using the $4 \times K$ A sub-block $A_{i..i+3,0..K-1}$ and the $K \times 4$ B sub-block $B_{0..K-1,j..j+3}$. For this `update4x4` operation, $C_{i..i+3,j..j+3}$ is loaded into 16 of the SPARC's FPU registers, and, the kth iteration, $0 \leq k < K$, a small "outer-product" update is applied to these registers, ie. a 4×1 segment of the A sub-block, $A_{i..i+3,k}$, is combined with a 1×4 segment the B sub-block, $B_{k,j..j+3}$. At the end of the K iterations, the updated register values are stored again in the C matrix.

This amounts to a 4×4 loop unrolling of (the outer indices of) an "inner-product" matrix multiply loop, which in turn enables *software pipelining* techniques to give the optimal ordering of the load, multiply and add instructions involved. For optimal performance, it was found that at least 2 floating point instructions need to separate a load instruction and a multiply instruction dependent on that load; 3 floating point instructions should separate a multiply instruction and an add instruction dependent on that multiply; and also at least 2 floating point instructions should separate a multiply instruction and the next load instruction modifying an operand of the multiply instruction (this is due to the fact that the load may otherwise be delayed until the multiply instruction is completed).

The ratio of arithmetic instruction to loads operations in this loop is (asymptotically) 4:1; on the SPARC IU, this amounts to a ratio of 4:1 for the cycles consumed in servicing FPU

K	16	32	48	64	80	96	112	128	160	192
speed	5.4	5.8	5.7	6.0	6.0	6.0	5.9	5.9	5.2	3.8

Table 2: Speed (MFLOPs) of `outerproduct` loop used to update a $K \times K$ matrix (for 16 repetitions) on a 1×1 AP1000

arithmetic to issuing FPU load instructions. This latter ratio was further increased to 16:3 by using the load double word instruction. This however in turn required A to be stored in transposed form, and A and B to have an even "last dimension". The results for this loop are given in Table 1; these are sustained figures using matrices A and B having at least half their elements in cache before the multiply begins. This was chosen to approximate the cache conditions for a multiplication over a larger AP1000, since part of A and B will already be in cache from message receipt. Performance decreases slightly for $K > 80$ due to slight cache conflict occurring. That the performance is better than 16/19 of the theoretical peak is due to the fact the the SPARC IU consumes 2/3 of the time to issue a floating point arithmetic instruction than does the FPU to execute it. Thus the `update4x4` loop is limited almost equally by SPARC IU and FPU performance, for the current AP1000 cell architecture.

For high K-factor matrices (eg. A is 32×256, B is 256×32) that are "pre-loaded" into cache, the `update4x4` loop operates at about 7.9 MFLOPs. This indicates the maximum performance of the loop itself, free from cache effects. An optimization for C being $N \times N$ gives a further 1% improvement.

Even with the optimization options available for the SPARC C compiler, comparable performance at present cannot be achieved using loops written in the C language. Originally, the `update4x4` loop was written in C, with local variables being mapped correctly to FPU registers, and assignment statements in a 1-1 correspondence with SPARC FPU instructions. This failed because the compiler tried to "optimize" the instructions by re-ordering them, destroying the software pipelining effect so that the performance went down to about 5.5 MFLOPs!

3.1.2 The `outerproduct` loop

An $N \times N$ matrix B may be updated by $1 \times N$ vectors a and b using $B \leftarrow B + a^T b$. This is called an "outer product" or "rank 1" update. For this `outerproduct` loop, B is grouped into rows (or columns) of 4, ie. $B_{i..i+3,0..N-1}$ ($B_{0..N-1,j..j+3}$), and the corresponding 1×4 segment of a (b), $a_{i..i+3}$ ($b_{j..j+3}$) is loaded into the FPU registers. For each k ($k = 0, 4, 8, \ldots, N-1$), a 4×4 sub-block of the B rows (columns), $B_{i..i+3,k..k+3}$ ($B_{k..k+3,j..j+3}$), and $b_{k..k+3}$ ($a_{k..k+3}$) are loaded into the FPU registers, a small outer product operation is performed, and the B sub-block is stored again.

Again this amounts to a 4×4 loop unrolling, with similar *software pipelining* techniques to the `update4x4` loop, with the additional constraint that 2 floating point operations should separate and add instruction and store instruction that is dependent on it.

The ratio of arithmetic instructions to load/store operations in this loop is (asymptotically) 8:9. On the SPARC IU, this amounts to a ratio of 8:11 for the cycles consumed in servicing FPU arithmetic to issuing FPU load/store instructions, which can be increased to 1:1 using load/store double word instructions. For the same reason as for `update4x4`, the performance is better than this ratio indicates, as indicated in Table 2. The performance of this loop is bound by the SPARC IU.

Originally, `outerproduct` loop was tested on the SPARC 1+, achieving only about 75% of the performance indicated above. The AP1000 cell's performance is superior since it has a "copy-back" cache, whereas the SPARC 1+ has a "write-through" cache, which causes delay due to cache flushing upon each store instruction.

3.1.3 Implications

Both loops were implemented as "leaf" procedure calls, ie. parameters are passed via SPARC output registers. This permitted negligible overhead for procedure calling, even for moderate matrix sizes.

Although `update4x4` is designed for the BLAS matrix multiplication routines and `outer-product` was designed for the BLAS matrix update routines, the former has clearly superior relative performance, which is likely to improve further with a modified AP1000 cell configuration. This means that effort should be taken to regroup and possibly even re-order the outer product computations of the BLAS-3 matrix update routines, so that `update4x4` can be used to perform as much as possible of the overall computation (Section 3.4).

One important consequence of the `update4x4` loop is that for a sufficiently large K, whether the result matrix C is in the cache is largely irrelevant, since a cache miss on accessing an element of C can occur at most twice, as opposed to $K/4$ times for each of A and B. Thus, the cache need only contain elements of A and B, and K be sufficiently large (ie. $K > 64$), for the loop to be able to achieve its maximum performance. This means that in parallel matrix multiplication, it is better to communicate A and B rather than A and C, as communicating the latter will force B out of the cache. It also assists the asymptotic efficiency of the partitioning methods (Section 3.2.3).

3.2 Parallel Matrix Multiply-Add Operations

In this section, parallel matrix multiply add operations, eg. $C \leftarrow C + AB$ where A, B, C are matrices distributed over the AP1000 cells, are considered. In [16], the "full-systolic" and "semi-systolic" methods were described.

A third parallel matrix multiplication method, called the "non-systolic" method, exploits the AP1000's fast `xy` broadcast routines.

> **for** (k=0; k $< N_x$; k++)
>   y-broadcast B cell sub-block from row k;
>   x-broadcast A cell sub-block from column k;
>   perform local cell sub-block multiplication;

This method must use explicit matrix transpose if one of the operands is transposed.

All three methods use the `xy` communication routines. Table 3 indicates the relative efficiency of each method. Here, the cell sub-block of the matrix B followed directly after, in main memory, that of A, to ensure they would map into separate areas of the AP1000's cell's direct-mapped cache [12], as far as possible.

For the 2×2 AP1000, the semi-systolic method was slightly faster for $K/N_x = 128$, being at 7.3MFLOPs/cell, due to the fact that it performs $2N_x - 1$ (as opposed to $2N_x$) communications. The table indicates that the full-systolic method, with its startup time to rotate A and B is the slowest, by a small margin; the difference between the other two methods is negligible, due to the fact that the `xy_send` and `xy` broadcast routines have virtually the same performance.

3.2.1 Explicit vs implicit matrix transposition

Most of the BLAS-3 routines require multiply-add operations in which the operands (A, B) may be transposed. As explained in [16], it is possible to avoid explicit matrix transposition if only one of the matrices is to be transposed using "systolic" or "semi-systolic" parallel multiply-add methods. It is not obvious whether this will improve efficiency as this means communicating the result matrix C (causing extra cache conflict if `update4x4` is used), whereas the fast AP1000 wormhole routing promises low relative overhead for the explicit transpose.

	$C \leftarrow C + AB$ by (-systolic) method:			$C \leftarrow C + A^T B$ by (-systolic) method:	
K/N_x	full-	semi-	non-	semi- (implicit)	non- (explicit)
16	4.2	4.4	4.4	4.3	4.1
32	5.8	6.0	6.0	5.9	5.8
64	6.5	6.7	6.8	6.7	6.7
96	7.0	6.9	7.0	6.9	6.9
128	7.1	7.2	7.2	7.1	7.1
160	7.1	7.2	7.1	7.1	7.1

Table 3: Speed in MFLOPs/cell of parallel multiply-add methods on an 8×8 AP1000 with $K \times K$ matrices

For explicit matrix transposition, the simplest method of just exchanging data between cells appears to be the most efficient. The bottleneck for this algorithm is at the diagonal cells, through each of which $N_x - 1$ messages must pass and change direction, so that the time is expected to be proportional to $N_x - 1$. This is verified by timings which yield a communication rate of 1.4MByte/s/cell for an 8×8 AP1000 ($64 \leq K/N_x \leq 256$); the timings also indicate that the relative overhead compared with matrix multiplication time is small for $K/N_x \geq 32$, reducing to about 0.5% for $K/N_x = 128$.

Table 3 indicates that for square matrices, there is little difference between the explicit and implicit methods, except for small matrices, which favour the implicit method.

3.2.2 Fast xy communication experiment

The `xy` communication routines are fast for moderate messages sizes, such as those indicated in Table 3. This is largely because the sending cell does not have to copy the message and that the parts of the message in cache can be sent more quickly using a "cache line send" function [15]. However, the receiving cell still has to copy its message from ring buffer to the user space, which, as well as consuming comparable time to the actual (DMA) message receive itself, further disturbs the cache.

For implementing libraries such as the BLAS-3, performance is more important than programming effort. It is desirable then to avoid the receiving cell message copying if possible, preferably by direct DMA receive into the user space. As it is not currently possible to do this using the current AP1000 CellOs, modified `x_brd()` and `y_brd()` routines were written so that these functions returned a pointer to the ring buffer where the message was directly received and the copy to user space avoided[1]. This method has the added bonus that the messages in the A and B sub-blocks, being received in the ring buffer one after the other, map into non-overlaping areas of the cache.

The performance of these variations was tested for the non-systolic multiply add method, and generally halved the communication overhead. Thus, for $K/N_x = 128$, performance of 7.5MFLOPs/cell was achieved, 90% of the theoretical peak.

3.2.3 Workspace and partitioning methods

As explained in [16], BLAS-3 routines operate on sub-blocks of larger matrices, rather than whole matrices as such. Using the scattered distribution strategy, these sub-blocks are therefore

[1] In the case of ring buffer "wrap around", the message would be still copied to user space. Since a large (512KB) ring buffer can be used on the AP1000, this point does not significantly affect efficiency.

A'_{0l}	B'_{lj}	A'_{1l}		A'_{2l}		$\dots$	$A'_{(m/k_0-1)l}$

Figure 1: The contents of the workspace for a partitioned multiply-add operation

generally not contiguous in memory when mapped to the AP1000 cells, which is inconvenient for both message passing and cache management. Furthermore, the matrix multiply-add operation also involves scaling by constants α and β. These problems can be most easily overcome by copying (parts of) the A, B and in some cases C sub-blocks into contiguous blocks in a BLAS-3 "workspace" area, which may then be scaled if needed[2].

This however is at odds with the philosophy of the authors of BLAS-3 [1], who see as undesirable any large (static) storage for BLAS-3 workspaces. On the other hand, the issue of workspace needs be re-considered for distributed memory machines, which generally have large aggregate memories, and temporary storage generally needs to be allocated for message transfer in any case.

Using dynamic storage allocation to create a workspace is a reasonable solution as extra storage is only required for the duration of a BLAS-3 procedure call. Furthermore, with the use of suitable partitioning methods on $N \times N$ matrices, an $O(N)$, or even an $O(1)$, workspace is sufficient for reasonable performance. An "outer product"-based $O(N)$ workspace partitioning method, capable of high asymptotic performance, will now be described.

Consider an $M \times K$ global matrix A having an $m \times k$ (sub-) matrix A' on a particular AP1000 cell, where $m = M/N_y, k = K/N_x$. Divide A' into $k_0 \times k_0$ sub-blocks denoted A'_{ij} where $0 \le i \le \lceil m/k_0 \rceil$, $0 \le j \le \lceil k/k_0 \rceil$ and the optimal block size $k_0 = 128$ is chosen from Table 3. Let B be a $K \times N$ global matrix divided up in a similar way.

The method involves at step l copying the "block-column" $A'_{0l}, A'_{1l}, \dots, A'_{(m-1)l}$ into a contiguous workspace, for $l = 0, \dots, k/k_0 - 1$. On the jth sub-step ($j = 0, \dots, n-1$), B'_{lj} is copied to the workspace and is multiplied by each of the k A' sub-blocks already there. The layout of these sub-blocks in the workspace is shown in Figure 1. Here, one can see that A'_{il} and B'_{lj} map into different areas of the AP1000's 128KB direct-mapped cache. For this reason, almost half of the workspace is unused. The total size of the workspace is $k_0(m + n - 1)$ words, and it can be seen that the cost of copying (with scaling, if needed) a sub-block into the workspace is amortised over the k/k_0 times it is used to perform a multiply-add.

With the `update4x4` loop as the core of the multiply-add operation, the position of the cell's result (sub-) matrix C' in cache is essentially irrelevant. This is fortunate as C' need never be copied into the workspace; if it had to, it would be very difficult to amortise the cost of the copying of A', B' and C' to and from the workspace simultaneously. Also, when an A' sub-block is copied to the workspace, it must transposed for the `update4x4` loop.

This idea can be efficiently integrated into the parallel non-systolic multiply-add method as follows (similarly for the semi-systolic method). If a cell is currently broadcasting its A' sub-matrix to other cells, for each sub-block of A', it copies the sub-block into the workspace and the broadcasts it from there (similarly if it is broadcasting B'). As sending a message is much faster than receiving one for the AP1000 `xy` communication routines, the copying of the next sub-block by the sending cell is overlapped with the receiving of the previous sub-block on the receiving cells. Although the message size of $4k_0^2$ bytes is convenient for the AP1000 ring buffer, eventually the sender cell gets well ahead of the receiving cells for very large matrices. This can cause ring buffer overflow unless the AP1000 cells are synchronized periodically.

The performance of this partitioning method are given in Table 4. As the maximum matrix size corresponds to 4MB, results for a 4×4 AP1000 are given; the results for an 8×8 appear

[2]While some authors have implemented distributed matrix multiplication without copying of the input matrices [14], this is not in general possible for the BLAS-3 as the input matrices can be the same (eg. $C \leftarrow AA^T$).

	partitioning	
N/N_x	yes	no
128	7.18	7.23
256	7.44	5.40
384	7.52	5.73
512	7.58	5.49
640	7.60	—
728	7.59	—
896	7.63	—
1024	7.65	—

Table 4: Speed in MFLOPs/cell of parallel $C \leftarrow C + AA$ using non-systolic method on an 4×4 AP1000 with $N \times N$ matrices

identical for the corresponding matrix sizes. These results indicate the performance achievable for the BLAS-3 general multiply operation $C \leftarrow \alpha AB + \beta C$, over 90% of the theoretical peak on the AP1000.

3.2.4 Implicit partitioning methods

For linear algebra algorithms to be efficiently implemented on architectures like the AP1000, the computational data must be divided into "blocks" that fit within the cache, and the computation must be reorganized so that as many operations as possible be performed on them while in cache. This is possible with matrix multiplication, as well as LU decomposition [3] and Singular Value Decomposition [4].

While the "explicit" partitioning method described above is entirely adequate for the BLAS-3, especially as it has consistent performance over varying matrix sizes, for other applications it might not be desirable to have to copy matrices to workspaces. An alternative is to use *implicit* partitioning methods which simply re-order the computations to make the best use of the cache. Such a scheme for the matrix multiplication $C \leftarrow C + AB$ on a single AP1000 cell will now be described.

Divide the $K \times N$ matrix B into $k_0 \times n_0$ sub-blocks, where $k_0 n_0$ does not exceed the cache size, and $K/k_0 > 1$. Now with B stored in row-major form and thus having ''last dimension" $\mathrm{ld}_B \geq N$ (assume for sake of simplicity that $\mathrm{ld}_B = N$), then such a sub-block B' will not be stored contiguously in main memory, and hence the rows of B' will map into non-contiguous areas in a direct-mapped cache.

Assume that A is an $N \times K$ matrix is stored in transposed form. The idea is that while B' is in the cache, take each appropriate $4 \times k_0$ "piece" of A and perform the `update4x4` operation $n_0/4$ times, on each of the $k_0 \times 4$ pieces of B'. During this time, *most* of B' and the "piece" of A' should be in cache, so that the performance should be high.

The underlying idea here is that only a large portion of one of the matrices (in this case B) need ever be in the cache at any one time. Certainly, a "piece" of A will sometimes map into the same area of cache as B', causing cache misses, but as the former is small, this happens sufficiently rarely to degrade performance seriously. It was found that with $k_0 = 128$, the optimal value of n_0 was 128. Table 5 summarizes the performance for $K = 128$ (the performance for $K > 128$ was very similar).

The results indicate that the implicit method achieved good cache utilization *except* where

	partitioning	
N	yes	no
128	7.56	7.57
256	7.55	5.54
384	7.63	6.80
512	5.51	5.51
640	7.64	6.47
728	7.60	6.46
896	7.63	6.47
1024	5.52	5.52

(a) for $128 \leq N \leq 1024$

	partitioning	
N	yes	no
384	7.63	6.80
416	7.38	6.71
448	7.13	6.60
480	7.61	6.53
512	5.51	5.51
544	7.63	6.80
576	6.99	6.47
608	7.61	6.70

(b) detail of $384 \leq N < 640$

Table 5: Speed in MFLOPs/cell of parallel $C \leftarrow C + AB$ using "implicit" partitioning on a 1×1 AP1000 with A being $N \times k_0$ and B being $k_0 \times N$ with $k_0 = n_0 = 128$

ld_B was a multiple of 512.[3] The reason for this is as follows. For $\mathrm{ld}_B = 128q$ and $n_0 = 128$, row i of B' can be thought as mapping into the (qi mod 256)th "row" of the cache, where the cache is conceptually divided into 256 rows of length n_0. If q is a power of 2, then for a sufficiently large k_0, rows i and $i + 2^{\log_2 k_0 - 1}$ must map into the same area of cache. Otherwise, each row of B' will map into different areas of cache, and the performance will be high[4]. For ld_B not being a multiple of 128, the performance will be between these two extremes, as here, only parts of the different rows of B' will map into the same area of cache.

It is possible to avoid this loss of performance by decreasing k_0 (or n_0). For example, with $k_0 = 64$, the multiplication for the 128×512 B runs at speed 7.45 MFLOPs, and for $k_0 = 32$, that for the 128×512 B runs at speed 7.18 MFLOPs.

3.3 Adaption to a rectangular AP1000 configuration and the BLAS-2 limit

We aim to implement a subset of the BLAS on a general rectangular AP 1000 configuration. We suppose that the configuration has N_y by N_x cells, and that matrices are stored using the scattered strategy described in Section 2.2. Preliminary results for matrix transpose and matrix multiply-add are described in Sections 3.3.1 and 3.3.2. In these Sections, the algorithms are also efficient on non-square matrix shapes. These shapes can approach (and reach) the BLAS-2 matrix-vector operations, hence the term "BLAS-2 limit".

3.3.1 Matrix transpose

The matrix transpose operation $B \leftarrow A^T$ is nontrivial if $N_x \neq N_y$. We have implemented the following algorithm (`ctranscz`). Consider the j-th column $A_{.j}$ of A, which is to be transposed to form the j-th row of B. The column $A_{.j}$ is distributed over cells with cidx $= j$ mod N_x, and the j-th row of B should be distributed over cells with cidy $= j$ mod N_y. Thus, each cell with cidx $= j$ mod N_x potentially has to send a message to all N_x cells with cidy $= j$ mod N_y, and each of these cells potentially has to receive a message from all N_y cells with cidx $= j$ mod N_x. In fact, because of the definition of the scattered strategy, the number of messages to be sent by each cell is at most N_x/G, and the number to be received is at most N_y/G, where $G = \mathrm{GCD}(N_x, N_y)$.

[3]These results do not take into account communication, so that they cannot be directly compared with Table 4.

[4]This situation is analogous to the sudden degradation of performance in interleaved memory vector processes, when the vector stride equals the number of interleaved memory banks.

configuration		time	speed
N_x	N_y	msec	Kbyte/sec/cell
1	64	250	250
2	32	99	631
3	21	81	786
4	16	81	771
5	12	127	525
6	10	91	736
7	9	123	518
8	8	48	1304

Table 6: 1000 by 1000 matrix transpose

A naive program would simply iterate over the column index j. However, this would waste most of the bandwidth available in the T-net, because only N_y cells would be sending and N_x receiving at any one time. It is more efficient to block several columns and send all messages associated with them at once. Our present implementation blocks up to $\max(N_x, N_y)$ columns, so there is at most one message from any cell (x, y) to any other cell (x', y'). We could block up to $\mathrm{LCM}(N_x, N_y)$ columns and retain this property, but this would increase the chance of ring buffer overflow (we assume that messages are sent using `xy_send`).

In order to attain reasonable efficiency, it is desirable for each cell to precompute permutations π_s and π_r, where π_s is associated with sending messages (part of a column of A), and π_r is associated with receiving messages (part of a row of B). More precisely, a sending cell has part (say v) of a column of A. Elements of v have to be sent to several other cells, but these elements are not stored in contiguous locations in v. The permutation π_s is defined so that $\pi_s v$ has all elements which are to be sent to a particular cell in contiguous locations. Similarly, a receiving cell receives several messages into adjacent blocks of memory, then has to "unscramble" them using the permutation π_r in order to obtain its share of a row of B in the locations defined by the scattered strategy.

A similar algorithm (`ctransrz`) transposes each row of A to a column of B. To minimise startup overheads, we choose `ctranscz` if A is M by K with $M > K$, and `ctransrz` if $M < K$.

Table 6 gives the time required (in msec) and the overall communication speed (in Kbyte/sec per cell) for 1000 by 1000 single-precision matrix transpose on various CAP configurations. It is apparent that, for matrices of this size, the time to form the transpose is much less than the time for matrix multiplication.

3.3.2 Matrix multiplication

Consider the matrix multiply-add operation $C \leftarrow C + AB$, where A is M by K, B is K by N, and C is M by N. We have implemented three algorithms, based on the "non-systolic" method of Section 3.2, and choose whichever is the most efficient. This depends, at least to a good approximation, on which of M, N and K is the smallest.

Method A performs K rank-1 updates to C, i.e. $C \leftarrow C + \sum A_{.j} B_{j.}$, where $A_{.j}$ is the j-th column of A and $B_{j.}$ is the j-th row of B. In a naive implementation, the cells with $\mathrm{cidx} = j \bmod N_x$ would broadcast $A_{.j}$ horizontally (using `x_brd`), the cells with $\mathrm{cidy} = j \bmod N_y$ would broadcast $B_{j.}$ vertically (using `y_brd`), each cell would perform a local rank-1 update, and this cycle would be repeated K times. However, it is better for each cell to accumulate a moderate number k of rows and columns and then perform a single rank-k update.

The problem with method A is that there are $2K$ startup overheads associated with the broadcasts of the K columns of A and K rows of B. The "non-systolic" algorithm described

Matrix dimensions			Configuration $N_x \times N_y$		
M	K	N	4×8	7×8	8×8
1	1000	1000	4.06	3.53	3.83
10	1000	1000	4.80	4.57	4.60
1000	1	1000	3.08	2.97	2.94
1000	10	1000	5.68	5.39	5.47
1000	1000	1	4.18	3.53	3.81
1000	1000	10	5.02	4.57	4.52
1000	1000	1000	6.20	6.43	6.77

Table 7: Speed (MFLOPs/cell) of matrix multiply-add

above for the case $N_x = N_y$ is faster because several columns (or rows) are broadcast at the same time. This is difficult to achieve if $N_x \neq N_y$.

Method B is designed for the case of small N. The matrix B is transposed (using the algorithm `ctranscz` described above), then each row of B^T is broadcast vertically (using `y_brd`). Each cell computes a local matrix-vector product, and then the vector results are summed horizontally (using a generalisation of `x_fsum`). The important point is that the (large) M by K matrix A remains in place, while only the (small) K by N matrix B and (small) M by N matrix C move between cells.

Method C is simply the dual of method B, and is efficient if M is small.

Similar methods can be developed for the "semi-systolic" multiply-add (Section 3.2). In this case, the `x_fsum` communication is avoided.

In Table 7 we give speeds in for the combination of methods A, B and C (`cmatmulz`) on three different configurations. The speed exceeds 50 percent of the theoretical peak speed (8.33 MFLOPs/cell) unless $\min(M, K, N) = 1$.

3.4 Inverted Triangular Matrix Update Operations

Consider multiplying a rectangular matrix B by the inverse of an $N \times N$ triangular matrix, A on a $N_x \times N_x$ AP1000. Firstly, we require the possible scaling of A (and hence B) so that A's diagonal is unit (in BLAS-3, it is up to the user to ensure that this is possible, ie. that A is non-singular).

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, $A_{.j}$; then, the "outer product" update $A \leftarrow A - A_{.j}A_{j.}$ is performed. Each cell communicates $O(n)$ data per $O(n^2)$ arithmetic operations, where $n = N/N_x$ 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 $A^{-1} = E_{1.} \dots E_{(N-1).}$. Thus, using this to form $B \leftarrow A^{-1}B$, we can perform the corresponding (parallel) row updates on B:

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

The algorithm is similar for the post-multiplication of A^{-1}, except that the columns of B and the rows of A are broadcast.

The non-inverted triangular update operation can be performed in a similar way, or, if the matrices are small enough, by a direct matrix multiply. Two implementations of triangular matrix updates are given below.

N/N_x	$A^{-1}B$			BA^{-1}		
	$N_x = 1$	$N_x = 2$	$N_x = 8$	$N_x = 1$	$N_x = 2$	$N_x = 8$
32	3.6	2.9		3.5	2.9	
64	4.8	4.0		5.0	4.1	
128	5.6	5.1	5.1	5.6	5.1	5.1
180	5.4	5.1		5.6	5.2	
256	4.4	4.1		3.0	2.8	

Table 8: Speed in MFLOPs/cell for $B \leftarrow A^{-1}B$, $B \leftarrow BA^{-1}$ for $N \times N$ matrices on the AP1000 (A is non-unit upper triangular)

3.4.1 Straightforward implementation

The results of directly implementing the above algorithm (with careful optimizations of all important pieces of the code) on the AP1000 are given in Table 8. The `outerproduct` loop is used to perform the update, and the efficiency of the basic algorithm without communication can be viewed from the $N_x = 1$ columns. For this algorithm, it is only important that B be kept in the cache; the performance for the BA^{-1} drops sharply as $N \to 256$ since the columns of B are being updated, rather than the rows. For the same reasons as those explained in Section 3.2.4, this increases cache sensitivity. It is difficult to design an effective partitioning strategy for this implementation.

The effect of communication on performance was much the same for $N_x > 2$ as $N_x = 2$. Communication overhead can be decreased further by using the fast versions of the `xy` broadcast routines, as described in Section 3.2.2. The performance does not quite approach that of the `outerproduct` inner loop; this is partly due to the scaling of a row/column of B with the corresponding diagonal element of A must occur directly before that row/columns is broadcast. With a unit diagonal A for $N/N_x = 128$, the speed increases to 5.8 MFLOPs for a 1×1 AP1000 and 5.2 MFLOPs for an 8×8 AP1000. Also, it is necessary to copy columns into contiguous messages for the `xy` routines. However, the main constraint on performance is the accumulated latency of many small broadcast operations, as well as the extensive use of the slower `outerproduct` loop.

3.4.2 Blocked implementation

Techniques for improving the LU decomposition algorithm on the AP1000 [3] can similarly be applied to the inverted triangular update algorithm. Consider the case of $A^{-1}B$, where A is $N \times N$ and upper triangular. The idea is to group ω row updates together, where $N_x|\omega$ and the blocking factor $\omega > 0$ is some function of N and N_x. Only the ω rows of B that are broadcast are updated using `outerproduct` loop (before their broadcast); these rows are grouped together into an $\omega \times N'/N_x$ matrix B'; similarly, the corresponding parts of the columns of A that were broadcast are "gathered" into an $N/N_x \times \omega$ matrix A' (the columns of A need not be broadcast individually, although the rows of B must be). Thus, the remaining N' columns of B (where $N' \leq N - \omega$) are updated with the product $A'B'$, so that the `update4x4` loop may be employed.

The performance of this version is given in Table 9 with the empirically found optimum $\omega = 2\sqrt{N_x N'}$. This formula represents a tradeoff between the proportion of work done in the `update4x4` (low ω) and a good K-factor (high ω) of the `update4x4` loop. Note that for $N/N_x \leq 360$, the performance is still improving; this is because A' and B' are relatively small matrices for this choice of ω, and hence can still fit in cache. For $N_x \geq 4$, speed improves at constant N/N_x, since the average K-factor increases with N. Although this is a considerable

N/N_x	$N_x=1$	$N_x=2$	$N_x=4$
32	3.6	3.4	3.5
64	5.3	5.0	5.2
128	6.3	6.1	6.3
180	6.5	6.4	6.4
256	6.6	6.6	6.7
360	6.8	6.9	6.7

Table 9: Speed in MFLOPs/cell for blocked $B \leftarrow A^{-1}B$ for $N \times N$ matrices on the AP1000 (A is non-unit upper triangular)

improvement over the unblocked version, performance is still below that of the `update4x4` loop, mainly because of the tradeoff mentioned above.

4 Future Research

As well as deciding for what matrix sizes and shapes the alternative algorithms of Section 3.3 should be employed for the "BLAS-2" limit, all algorithms presented here need to be adapted to a non-square AP1000 configuration.

Secondly, an implementation and evaluation of BLACS [11] on the AP1000 needs to be undertaken; at this point, the blocked panel-wrapped matrix distribution scheme needs to be addressed. When a consensus on the external interface of the BLAS-3 on a distributed memory architecture is reached, this will have to be built into the AP1000 implementation. This will complete the research component of this project.

Finally, it is hoped that a production level implementation of the BLAS-3 be made available; however, the programming effort involved is considerable. Double precision real and complex versions of the BLAS-3 will be required for realistic scientific computations. Then the relative performance and programming effort of linear algebra algorithms using the BLAS-3 with those not using it can be compared. For future versions of the AP1000, the BLAS-3 software will also need some maintenance to keep its performance optimal.

5 Conclusions

Conclusions are now given with respect to three of this project's major aims.

5.1 Usefulness of techniques used for BLAS-3 for other applications

As the BLAS-3 comprise relatively simple and easy to visualize matrix operations, the techniques used to obtain efficiency here may be applied to other areas (see [4]). For example, the design of the inner loops described in Section 3.1, especially with respect to the reuse of data using the FPU registers, need to be applied to any linear algebra algorithm's inner loops, in order to get optimal efficiency. On a higher level, regrouping computations into larger blocks is a necessary technique for almost any efficient parallel implementation, but the techniques are largely algorithm-dependent.

Once blocking is achieved, partitioning methods need to be employed; the "outer-product" partitioning using workspaces generally corresponds to a partitioning method cutting across the algorithm's innermost loop. This method is relatively easy to implement, has consistent performance, but has the cost of copying data to a workspace. The "implicit" method has different properties; generally, where an efficient partitioning scheme can be employed, either of these techniques can be applied.

Finally, "systolic", "semi-systolic" and "non-systolic" versions of many linear algebra algorithms exist; the results presented above indicate that, all other things being equal, the efficiency of the AP1000's cell broadcast routines favour the latter choices, but also that as AP1000 communication is relatively fast, optimizing communication is less urgent on the AP1000 than on comparable architectures.

5.2 Suitability of the BLAS-3 for distributed memory architectures

This project can address this issue from the implementor's point of view.

A major software engineering effort is required to get a robust implementation for the BLAS-3 yielding high performance over all the matrix sizes, shapes and orientations possible. Furthermore, it is fairly important to keep the code size bounded so as not to waste cell memory space. For example, all 4 orientations for the 4 types of triangular matrices have been coded for the inverted triangular update implementation described in Section 3.4.1. Considerable effort had to be invested to keep the code size compact and yet efficient.

To keep code size manageable, we have designed the BLAS-3 in terms of libraries of more primitive operations: parallel, AP1000-dependent matrix operations (eg. matrix transpose, non-systolic matrix multiply-add operations), which are themselves designed in terms of of a library of serial AP1000-independent matrix operations (eg. scale a matrix). These more primitive procedures have been optimized according to their importance.

This question remains to be asked: is a set of more primitive parallel matrix operations more appropriate than the BLAS-3 for distributed memory architectures? At least it must be asked whether all of the matrix orientations (especially the transposes) offered by the BLAS-3 are really necessary. The answers are beyond the scope of this paper, but in defense of the BLAS-3, a great deal more programming effort is required to implement *any* linear algebra algorithm on a distributed memory machine than on a shared memory machine, so the need for BLAS-3 becomes even more important, justifying the extra effort.

5.3 Suitability of the AP1000 for the BLAS-3

In this section, the suitability for the BLAS-3 (and hence numeric computation in general) of various features of the AP1000 are summarized.

The FPU. The FPU currently used in the AP1000 is its main weakness as a numeric machine, since its peak performance of 8.3 MFLOPs (single precision) is no longer exceptional in state-of-the-art processor technology. AP1000 cells need more and/or faster and/or pipelined FPUs. To take advantage of this, the SPARC IU will at least need a faster rate of issuing FPU instructions[5], as at present the `outerproduct` and to some extent even the `update4x4` loops' performances are IU bound. The other aspects of the AP1000 are sufficiently balanced to be able support this kind of improvement, and the advantage of basing the AP1000 cells on the SPARC architecture makes such improvements relatively easy.

The cache. The large, direct-mapped, copy back cache of the AP1000 is well-suited for high performance numeric computations. The copy back method is important for the performance of loops such as the `outerproduct` loop. The direct-mapped cacheing strategy, although requiring more programming effort to achieve full utilization, is better than a smaller or slower cache using a more sophisticated strategy.

Cell memory. The AP1000's 16MB cell memory is very important as users generally want to run computations on very large data sets on high performance supercomputers. To make

[5] Currently, the SPARC IU appears to require 2 cycles to do this, but presumably this could be reduced to one cycle with a modification in the cell architecture.

best use of cache and to amortize communication costs, considerable workspace area can be made available. Finally, as explained in [16], the aggregate cell memories, rather than the host's memory, should be regarded as the "ultimate" storage area for AP1000 computations on large sets of data.

Communication. The fast `xy` communication routines using the wormhole routed T-Network, especially the efficient broadcast routines, are one of the best features of the AP1000, enabling sustained performance very close to peak performance. Although originally designed for small messages, this project has shown that they can be used, with a little care, for communicating large amounts of data. A useful addition would be variants of the `xy` routines that performed DMA receive directly into a prescribed user buffer area. Although this may present semantic difficulties and dangers to the programmer, these are not serious problems for numeric computations which are typically single-process, and especially for the BLAS-3 which should be implemented by non-naive users. To a lesser extent, the software implementation of stride DMA send and receive [15] would also be useful.

Overall, the AP1000 is capable of achieving 80%-90% of its theoretical peak speed for the BLAS-3 routines for a wide range of matrix sizes. Very few parallel supercomputers can achieve such a high percentage, and this indicates that the AP1000 is a well balanced architecture. As future versions of the AP1000 scale upwards in computational performance, we expect it will remain a competitive cost-effective numerical supercomputer.

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, "The LINPACK Benchmark on the AP1000: Preliminary Report", Proc. *CAP Workshop 91*, Australian National University, 1991.

[4] A. Czezowski and P. E. Strazdins, "Singular value computation on the AP1000", Proc. *CAP Workshop 91*, Australian National University, 1991.

[5] 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.

[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] J. J. Dongarra, *LAPACK Working Note 34: Workshop on the BLACS*, Technical Report CS-91-134, University of Tennessee, 1991.

[12] H. Ishihata, T. Horie, S. Inano, T. Shimizu and S. Kato, "CAP-II Architecture", Proc. *First Fujitsu-ANU CAP Workshop*, Fujitsu Laboratories, Kawasaki, Japan, 1990.

[13] M. Ikesaka and T. Horie "Software Environment of CAP-II", Proc. *First Fujitsu-ANU CAP Workshop*, Fujitsu Laboratories, Kawasaki, Japan, 1990.

[14] S. L. Johnsson, T. Harris and K. K. Mathur, *Matrix Multiplication on the Connection Machine*, Technical Report NA89-3, Thinking Machines Corporation, Cambridge MA, 1989.

[15] T. Shimizu, H. Ishihata, and T. Horie, "Performance Evaluation of Message Transfer on CAP-II", Proc. *First Fujitsu-ANU CAP Workshop*, Fujitsu Laboratories, Kawasaki, Japan, 1990.

[16] P. E. Strazdins and R. P. Brent, "Implementation of the BLAS level 3 on the CAP-II", Proc. *First Fujitsu-ANU CAP Workshop*, Fujitsu Laboratories, Kawasaki, Japan, 1990.