Implementation of the BLAS Level 3 and LINPACK benchmark on the AP1000

Richard P. Brent and Peter E. Strazdins,
Computer Sciences Laboratory and Department of Computer Science,
Australian National University,
Canberra 2601 Australia
rpb@cslab.anu.edu.au, peter@cs.anu.edu.au

Abstract

This paper describes an implementation of Level 3 of the Basic Linear Algebra Subprogram (BLAS-3) library and the LINPACK Benchmark on the Fujitsu AP1000. The performance of these applications is regarded as important for distributed memory architectures such as the AP1000. We discuss the techniques involved in optimizing these applications without significantly sacrificing numerical stability. Many of these techniques may also be applied to other numerical applications. They include the use of software pipelining and loop unrolling to optimize scalar processor computation, the utilization of fast communication primitives on the AP1000 (particularly row and column broadcasting using wormhole routing), blocking and partitioning methods, and 'fast' algorithms (using reduced floating point operations). These techniques enable a performance of 85-90% of the AP1000's theoretical peak speed for the BLAS Level 3 procedures and up to 80% for the LINPACK benchmark.

1 Introduction

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) should be used where possible.

The LINPACK Benchmark involves the solution of a nonsingular system of n linear equations in n unknowns, using Gaussian elimination with partial pivoting and double-precision (64-bit) floating-point arithmetic. The performance of the LINPACK Benchmark and the BLAS-3 are both regarded as good indicators of a parallel computer's potential in numeric applications.

The AP1000 [2, 3] is a distributed memory MIMD machine with up to 1024 independent SPARC processors which are called *cells*, connected via a toroidal topology using wormhole routing. Each processor has a 128KB direct-mapped copy-back cache, 16MB of memory and a FPU of theoretical peak of 8.3 MFLOPs (single precision) and 5.6 MFLOPs (double precision). Details of the AP1000 architecture and software environment are discussed elsewhere in this issue.

High level design issues, most importantly the distribution of matrices over the AP1000, are discussed in Section 2. Techniques for the optimization of matrix computations on single AP1000 cells are given in Section 3. Section 4 describes the implementation of parallel matrix multiply-add operations on the AP1000, discussing issues such as communication, cache, non-square matrix shapes, and so-called 'fast' multiplication methods. The implementation of the

LINPACK Benchmark is discussed in Section 5, emphasizing the need for 'blocking' together small computations into larger ones. The application of this and techniques from Section 4 to the similar problem of BLAS-3 triangular matrix update is given in Section 5.3. Conclusions are given in Section 6.

1.1 The BLAS Level 3

The BLAS Level 3 [4] implement matrix-matrix operations, which, for $n \times n$ 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) [5], although degenerate cases of the BLAS-3 routines yield all BLAS-2 routines. The use of BLAS-3 is attractive on parallel machines such as the AP1000 because the cost of a data transfer may be amortized over the cost of $O(n)$ arithmetic operations.

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 of 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} or A^{-T}. Matrices may be general rectangular, symmetric or triangular but there is no special form of "packed" storage for symmetric or triangular matrices.

1.2 The LINPACK Benchmark

The LINPACK Benchmark involves the solution of a nonsingular system of n linear equations in n unknowns, using Gaussian elimination with partial pivoting and double-precision (64-bit) floating-point arithmetic. Three cases are considered:

1. $n = 100$ – the original benchmark.
2. $n = 1000$ – gives more opportunity for vector pipeline machines (and to some extent parallel machines) to exhibit high performance.
3. n as large as desired – gives maximum opportunity for vector pipeline and parallel machines to exhibit high performance.

We are only concerned with the cases 2 and 3 here, since case 1 is trivial to solve on a machine as powerful as the AP1000.

1.3 Conventions and Restrictions

We use the C language for implementation, as it permits better access to the low-level details of the AP1000, which is useful for optimizations. Thus, we use C conventions for matrices, stored in row-major ordering with indices starting from 0. 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 $m <= m'$. Let $A_{i.}$ denote the ith row and $A_{.j}$ denote the jth column of the matrix A.

Let N_x (N_y) be the number of cells across a row (column) of an AP1000 configuration, and $P = N_y N_x$ be the total number of processors. Our algorithms are will be first described for a square ($N_x \times N_x$) AP1000, and then generalizations to other AP1000 configuration will be given. A minor restriction is that for an $m \times n$ matrix to be distributed over the AP1000, we must have $N_y | m$ and $N_x | n$ (if necessary, matrices can be padded with 'dummy' rows and columns to satisfy this restriction).

2 High-level Design Issues

On non-distributed memory machines, calls to the BLAS-3 and LINPACK routines reference global matrices; to achieve the same effect on a distributed memory machine, we must have all AP1000 cells calling, in SPMD mode, the corresponding routine with references to the cell's respective sub-matrix of the global matrix. This unfortunately means that uniprocessor codes involving these routines cannot be directly ported to AP1000 cell programs.

To consider the optimal matrix distribution strategy, let us first consider what communication patterns are needed for these applications. These include, most importantly, (grouped) row/column broadcasts, row/column send/receive (for pivot row interchange for LINPACK and matrix rotation for 'systolic' matrix multiply) and finally row/column scan (eg. vector maximum for LINPACK).

For reasons of symmetry, high bandwidth for the row and column broadcasts, and good load balancing (especially for operation on triangular matrices and contiguous sub-blocks of larger global 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.

A generalization of all these distribution strategies is the 'blocked panel-wrapped' strategy, which is sufficient for all dense linear algebra applications in practice [6]. We have not implemented our algorithms for this more general strategy, as it introduces considerable coding difficulties. Also, due to the the relatively low communication startup overheads on the AP1000, it would not yield significantly better performance.

3 Optimizing Computation on SPARC processors

To optimize floating point computation on AP1000 cells, we have implemented kernels which are essentially a subset of uniprocessor BLAS-2 and BLAS-3 routines, optimized for the SPARC architecture used in AP1000 cells and written in SPARC assembly language. For this purpose, the following techniques were used:

1. write SPARC "leaf" routines to minimize procedure call overheads [16].

2. keep all variables and array elements in registers, to re-use as much as possible; this enables a low load/store to floating point operation ratio (denoted R).

3. use *software pipelining*, ie. separate loads, multiplies, adds, and stores which depend on each other by a sufficiently large number of instructions so that their operands are always available when needed.

Techniques 2 and 3 were achieved by using typically a 4×4 (for single precision) and a 4×2 (for double precision) loop unrolling.

The most important of such kernels was the Level 3 `UpdateRect()` routine which, for single precision, involves a matrix multiply-add $C \leftarrow C + AB$ where A is $4 \times k$ and B is $k \times 4$. This routine would initially load C into the FPU registers, and, upon the ith iteration, update it using $A_{.i}B_{i.}$, $0 \leq i < k$.

`UpdateRect()` has $R = 0.375$ (double precision) and $R = 0.25$ (single precision); the latter can be effectively reduced further to $R = 1/6$ using the SPARC load double word instruction. When used to perform an $n \times n$ matrix multiplication (with a 'warm' cache), `UpdateRect()` can sustain 7.7 MFLOPs ($80 \leq n \leq 160$) for single precision, and 5.1 MFLOPs ($56 \leq n \leq 120$) for double precision.

The next most important kernel is the Level 2 `Rank1Update()`, which implements the multiply-add $C \leftarrow C + ab$ where a is $m \times 1$ and b is $1 \times n$. A naive implementation would have $R = 1.5$; however, for single precision, using a 4×4 loop unrolling, this can be reduced to 1.125, again effectively reducible to slightly less than unity using the load double word instruction. `Rank1Update()` can sustain 5.9 MFLOPs ($64 \leq n \leq 128$) for single precision, and 4.0 MFLOPs ($48 \leq n \leq 100$) for double precision.

The other Level 2 routines, vector-matrix multiply and matrix-vector multiply, can sustain 7.3 MFLOPs (single precision) and 5.0 MFLOPs (double precision) for matrix multiplication.

These routines can achieve about the same percentage of the theoretical peak on the SPARC Station 1+ and SPARC 2 processors. An exception is `Rank1Update()`, which operates about 25% slower on these architectures, due to their 'write-through' caches.

The implications of these results for the following sections are as follows:

- use `UpdateRect()` wherever possible, even if it requires re-organization of the algorithm.
- using `UpdateRect()` to perform $C \leftarrow C + AB$ means that only A and B are significant with respect to the cache. This make good cache utilization much easier. For parallel algorithms, it is better to chose an algorithm not involving the communication of C, as message receipt of C may then displace either A or B from the cache.

4 Implementing BLAS-3 Parallel Matrix Multiply-Adds

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, firstly for an $N_x \times N_x$ AP1000, and then for a general AP1000 configuration (Section 4.4). The simplest parallel matrix multiplication algorithm, which we call the 'non-systolic' method, is as follows:

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;

A variation is the 'semi-systolic' method [14] where B cell sub-blocks are broadcast from the kth diagonal (instead of from the kth row), and each A cell sub-block is shifted right one unit (instead of broadcast). A third variation is the 'full-systolic' method (also known as Cannon's algorithm) in which both A and B sub-blocks are rotated at each step; this however has the overhead that both A and B must be initially 'aligned'.

Table 1 indicates the relative efficiency of each method for single precision. The overhead of the initial matrix alignment of the 'full-systolic' method makes it the slowest.

n/N_x	$C \leftarrow C + AB$ by (-systolic) method:			$C \leftarrow C + A^T B$ by (-systolic) method:	
	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 1: Speed in MFLOPs/cell of parallel multiply-add methods on an 8×8 AP1000 with $n \times n$ matrices (single precision)

To compute $C \leftarrow C + A^T B$ without using explicit transposition, variations of the 'semi-systolic' and the 'systolic' methods can be used where C's cell sub-blocks are communicated in place of those of B (similarly for $C \leftarrow C + AB^T$). This has a small overhead in extra disturbance of the cache, as explained in Section 3.

For explicit matrix transposition, the simplest method of exchanging matrix sub-blocks 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 $N_x - 1$ (for constant n/N_x). Transposition has a communication rate of 1.4MB s^{-1} per cell for an 8×8 AP1000 ($64 \leq n/N_x \leq 256$), which implies a small relative overhead (for $n/N_x \approx 128$, the overhead is about 0.5%).

Table 1 indicates that for square matrices, there is little difference between the explicit and implicit methods, except for small matrices, which favour the implicit method. This is due to the high relative speed of the AP1000 communication routines, which make the choice of communication patterns less critical.

4.1 Effect of Communication

Comparison of Table 1 with the results of Section 3 shows that the effect of communication on performance is appreciable, at least for moderate matrix sizes.

In the AP1000's `xy` communication routines, copying of matrices is avoided on message send; however, upon message receipt, messages are copied from a 'ring buffer' to user space. Message copying creates a twofold overhead, since message transfer (in hardware) on the AP1000 is almost as fast as a corresponding memory transfer, and message copying may disturb the cache. We made slight modifications to the `xy` routines so that the A and B cell sub-blocks were accessed directly from the ring buffer, thus avoiding the copy.

The performance of this optimization was tested for the non-systolic multiply-add method, and generally halved the communication overhead. Thus, for $n/N_x = 128$, a performance of 7.5 MFLOPs/cell (single precision) was achieved, 90% of the theoretical peak.

4.2 Optimizing Cache Utilization and Partitioning

BLAS-3 routines generally operate on sub-blocks of larger matrices, rather than whole matrices as such. Using the scattered distribution strategy, these sub-blocks are generally not contiguous in memory when mapped to the AP1000 cells, which is inconvenient for both message passing

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

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

	partitioning		Strassen's
n/N_x	yes	no	method
128	7.18	7.23	7.2 (7.2)
256	7.44	5.4	6.9 (7.9)
384	7.52	5.7	—
512	7.58	5.5	6.8 (8.8)
640	7.60	—	—
728	7.59	—	—
896	7.63	—	—
1024	7.65	—	6.7 (10.0)

Table 2: Speed in MFLOPs/cell of parallel $C \leftarrow C + AA$ using the non-systolic method on an 4×4 AP1000 with $n \times n$ matrices (single precision)

and cache management. Furthermore, the matrix multiply-add operation may involve scaling by constants α and β. Finally, distributed implementations of the BLAS-3 $C \leftarrow \alpha AA^T + \beta C$ imply copying of A cell sub-blocks, even if $\alpha = 1$.

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, where they may then be scaled if necessary. However, the workspace need not be $O(n^2)$ for $n \times n$ matrices; below we present an 'outer product'-based $O(n)$ workspace partitioning method, capable of high asymptotic performance by full utilization of the cache.

Consider an $m \times n$ global matrix A having an $m' \times k'$ (sub-) matrix A' on a particular AP1000 cell, where $m' = m/N_y, k' = k/N_x$. Partition A' into $k_0 \times k_0$ sub-blocks denoted A'_{ij} where $0 \leq i \leq \lceil m'/k_0 \rceil$, $0 \leq j \leq \lceil k'/k_0 \rceil$ and the optimal block size $k_0 = 128$ (for single precision) is chosen from Table 1. Let B be a $k \times n$ global matrix partitioned in a similar way.

The method involves at step l copying the 'block-column' $A'_{0l}, A'_{1l}, \ldots, A'_{(m'-1)l}$ into a contiguous workspace, for $l = 0, \ldots, k'/k_0 - 1$. On the jth sub-step ($j = 0, \ldots, n' - 1$), B'_{lj} is copied to the workspace and is multiplied by each of the k/k_0 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 per cell, and it can be seen that the cost of copying (with scaling, if needed) a sub-block into the workspace is amortized over the k/k_0 times it is used to perform a multiply-add.

This idea can easily be integrated into the parallel 'non-systolic' multiply-add, thus amortizing communication costs. The performance of this partitioning method is given in Table 2. As the maximum matrix size corresponds to 4MB, results for a 4×4 AP1000 are given; however the results for an 8×8 AP1000 appear 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.

It is possible to use partitioning without workspaces, where the overall matrix multiply is split into a series of sub-multiplications that minimize cache conflicts [15]. However, with a direct-mapped cache this cannot always yield maximum performance (eg. a $k \times n$ matrix B with $k\mathrm{ld}_B$ exceeding the cache size will mean that some elements in a single column of B will map into the same place in the cache).

4.3 'Fast' Methods

The above implementations are all based on standard ($O(n^3)$) matrix multiplication algorithms; however, with an 'acceptable' loss of numerical stability (in terms of the BLAS-3 error bounds [8]), it is possible to implement matrix multiply algorithms with a reduced number of arithmetic operations. One such algorithm, Strassen's method [8], has asymptotically $O(n^{2.81})$ operations.

In Strassen's method, matrices are split into 4 sub-matrices; products of the sums and differences of these sub-matrices may be combined in such a way that only 7 (instead of 8) sub-matrix multiplies need be computed. Thus, considerable workspace area is needed. If the matrix dimensions are powers of 2, this process can be easily repeated recursively. However, for $n \times n$ matrices, we have found it more efficient to apply only the first $\log_2(n/(N_x k_0))$ stages of the method, where k_0 is defined in Section 4.2, and hence it is only appropriate for large matrices. Table 2 gives the results of our implementation; in parentheses are the MFLOPs rating if $2n^3$ arithmetic operations are assumed. The actual efficiency decreases primarily because the FPU can operate at no more than half speed during the matrix addition and subtraction operations.

4.4 Adaption to a General AP1000 Configuration and the BLAS Level 2 Limit

We now describe an implementation of $C \leftarrow C + AB$, where C is $m \times n$, A is $m \times k$ and B is $k \times n$, for a general $N_y \times N_x$ AP1000 configuration; this implementation is also efficient in the cases where a matrix becomes a vector, hence the term 'BLAS Level 2 limit'.

In these cases, it is important to communicate the smaller of the matrices, so as to reduce communication costs. This may require transposition of the matrix beforehand (cf. the implicit transpose operations of Section 4). An efficient matrix transpose operation $A' \leftarrow A^T$ is nontrivial if $N_x \neq N_y$, and involves blocking and permuting matrix segments [15]. Our implementation, for a 1000×1000 matrix, achieves speeds on a 4×8, 7×8 and 8×8 configurations of (respectively) 1.02, 0.59 and 1.30 MB s^{-1} per cell.

The following three algorithms, based on the 'non-systolic' multiply-add of Section 4, are each suited to particular matrix shape:

- **A** (for small k) perform k rank-1 updates to C, ie. $C \leftarrow C + \sum A_{.j} B_{j.}$. The cells in column $j \bmod N_x$ of the AP1000 broadcast $A_{.j}$ horizontally, the cells in row $j \bmod N_y$ of the AP1000 broadcast $B_{j.}$ vertically. Each cell accumulates a moderate number ω of these broadcasts and then performs a single rank-ω update. The $2k$ broadcast startup overheads involved here can be reduced by grouping if $\mathrm{GCD}(N_y, N_x) > 1$.
- **B** (for small n) transpose B, then broadcast each row of B^T. Each cell computes a local matrix-vector product, and the vector results are summed horizontally.
- **C** (for small m) is simply the dual of **B**.

In Table 3 we give speeds for the combination of methods **A**, **B** and **C** on three different configurations. The speed exceeds 50 percent of the theoretical peak speed (8.33 MFLOPs/cell) except for the case $\min(m, n, k) = 1$.

m	k	n	4×8	7×8	8×8
1	1000	1000	4.1	3.5	3.8
10	1000	1000	4.8	4.6	4.6
1000	1	1000	3.1	3.0	2.9
1000	10	1000	5.7	5.4	5.5
1000	1000	1	4.2	3.5	3.8
1000	1000	10	5.0	4.6	4.5
1000	1000	1000	6.2	6.4	6.8

Table 3: Speed (MFLOPs/cell) of matrix multiply-add on rectangular AP1000 configurations (single precision)

5 Implementing the LINPACK Benchmark

Suppose we want to solve a nonsingular n by n linear system:

$$Ax = b \tag{5.1}$$

on an $N_x \times N_x$ AP1000. The augmented matrix $[A|b]$ is stored using the scattered representation.

It is known [11, 13] that Gaussian elimination is equivalent to triangular factorization. More precisely, Gaussian elimination with partial pivoting produces an upper triangular matrix U and a lower triangular matrix L (with unit diagonal) such that:

$$PA = LU \tag{5.2}$$

where P is a permutation matrix. In the usual implementation A is overwritten by L and U (the diagonal of L need not be stored). If the same procedure is applied to the augmented matrix $\bar{A} = [A|b]$, we obtain

$$P\bar{A} = L\bar{U} \tag{5.3}$$

where $\bar{U} = [U|\bar{b}]$ and (5.1) has been transformed into the upper triangular system

$$Ux = \bar{b} \tag{5.4}$$

In the following we shall only consider the transformation of A to U, as the transformation of b to $\bar{b}$ is similar.

If A has n rows, the following steps have to be repeated $n-1$ times, where the kth iteration completes the computation of the kth column of U:

1. Find the index of the next pivot row by finding an element of maximal absolute value in the current (kth) column, considering only elements on and below the diagonal.

2. Broadcast the pivot row vertically.

3. Exchange the pivot row with the kth row, and keep a record of the row permutation.

4. Compute the "multipliers" (elements of L) from the kth column and broadcast horizontally.

5. Perform Gaussian elimination (a rank-1 update using the portion of the pivot row and the other rows held in each cell).

We can estimate the parallel time T_P involved:

$$T_P \simeq \alpha n^3/N_x^2 + \beta n^2/N_x + \gamma n, \tag{5.5}$$

where the first term is due to the $2n^3/3+O(n^2)$ floating point operations, the second term is due to the total volume of communication, and the third due to the communication startup (eg. $O(n)$ row/column broadcasts). The terms are additive as it is difficult to overlap computation with the AP1000's `xy` communication. As we would expect the time on a single cell to be $T_1 \simeq \alpha n^3$, the efficiency E_P is:

$$E_P \simeq \frac{1}{1+(1+\bar{\gamma}/n')\bar{\beta}/n'} \simeq \frac{1}{1+n_{half}/n}, \tag{5.6}$$

where $\bar{\beta} = \beta/\alpha$ is proportional to the ratio of communication to computation speed, $\bar{\gamma} = \gamma/\beta$ measures the importance of the communication startup time, $n' = n/N_x$, and $n_{half} = \bar{\beta} N_x$ is the problem size giving efficiency 0.5 (this approximation is valid if $\bar{\gamma}$ is negligible). From (5.6), the efficiency is close to 1 only if $n' \gg \bar{\beta}$.

We omit details here of the "back-substitution" phase, ie. the solution of the upper triangular system (5.4), because this can be performed in time much less than (5.5) (see [9, 12]). For example, with $n = 1000$ on an 8×8 AP1000, the back-substitution phase takes 0.1s as opposed to the LU factorization phase, which takes 3.5s. A generalization of the back-substitution phase (with the vector b becoming a matrix) will be discussed in Section 5.3.

To adapt this algorithm to an $N_y \times N_x$ AP1000 with $N_y = 2N_x$, our *ad hoc* solution was to simulate a $N_y \times N_y$ AP1000 by each physical AP1000 cell simulating two virtual cells in the x-direction. This ensured full processor utilization and optimal communication speed, but due to the significant costs of context switching on AP1000 cells, the simulation was hard coded rather than using two tasks per cell.

5.1 The Need for Blocking

As discussed in Section 3, peak performance cannot be reached using rank-1 updates. It is possible to reformulate Gaussian elimination so that most of the floating-point arithmetic is performed in matrix-matrix multiplications, without compromising the error analysis. Partial pivoting introduces some difficulties, but they are surmountable. The idea is to introduce a "blocksize" or "bandwidth" parameter ω. Gaussian elimination is performed via rank-1 updates in vertical strips of width ω. Once ω pivots have been chosen, a horizontal strip of height ω can be updated. At this point, a matrix-matrix multiplication can be used to update the lower right corner of A. The optimal choice of ω is best determined by experiment, but

$$\omega \simeq n^{1/2}$$

is a reasonable choice, with ω a multiple of N_x.

Here, we take advantage of each AP1000 cell's relatively large memory (16 MB) and save the relevant part of each pivot row and multiplier column as it is broadcast during the horizontal and vertical strip updates. The block update step can then be performed independently in each cell, without any further communication. Each cell requires working storage of about $2\omega n/N_x$ floating-point words, in addition to the $(n^2 + O(n))/N_x^2$ words required for the cell's share of the augmented matrix and the triangular factors. If $2\omega n/N_x$ exceeds the cache size, partitioning methods for the matrix multiply need to be employed (see Section 4.2).

The effect of blocking is to reduce the constant α in (5.5) at the expense of increasing the lower-order terms. Thus, a blocked implementation should be faster for sufficiently large n, but may be slower than an unblocked implementation for small n. This is what we observed – with our implementation the crossover occurs at $n \simeq 40N_x$.

Time for one cell	cells	time (sec)	speedup	efficiency
160	512	1.10	147	0.29
160	256	1.50	108	0.42
160	128	2.42	66.5	0.52
160	64	3.51	46.0	0.72
160	32	6.71	24.0	0.75
160	16	11.5	13.9	0.87
160	8	22.6	7.12	0.89
160	4	41.3	3.90	0.97
160	2	81.4	1.98	0.99

Table 4: LINPACK Benchmark results for $n = 1000$

5.2 Results

The benchmark programs perform Gaussian elimination with partial pivoting (and check the size of the residual). All results are for double-precision. Single-precision is about 50 percent faster.

As discussed in Table 3 of [1], a gain in efficiency of up to 40% is achieved by blocking over non-blocking for large matrices. Also, a version the blocked algorithm was implemented where the AP1000's hardware-supported row/column broadcast and scan operations were simulated in software. This version ran 7% slower even for large matrices, indicating the need for hardware support for these operations.

The results in Table 4 are for $n = 1000$ and should be compared with those in Table 2 of [7]. The results in Table 5 are for n almost as large as possible (constrained by the storage of 16 MB/cell), and should be compared with those in Table 3 of [7]. In Table 5:

n_{max} is the problem size giving the best performance r_{max},

n_{half} is the problem size giving performance $r_{max}/2$, and

r_{peak} is the theoretical peak performance (ignoring everything but the speed of the floating-point units).

The results for the AP1000 are good when compared with reported results for other distributed memory MIMD machines such as the nCUBE, Intel iPSC/860, and Intel Delta, if allowance is made for the different theoretical peak speeds. For example, the 1024-cell nCUBE 2 achieves 2.59 sec for $n = 1000$ and 1.91 GFLOPs for $n = 21376$ [7] with $r_{peak} = 2.4$ GFLOPs. Our results indicate that a P-cell AP1000 is consistently faster than a $2P$-cell nCUBE 2. The 512-cell Intel Delta achieves 13.9 GFLOPs but this is less than 70 percent of its theoretical peak of 20 GFLOPs [10]. The 128-cell Intel iPSC/860 achieves 2.6 GFLOPs, slightly more than the 512-cell CAP, but this is only 52 percent of its theoretical peak of 5 GFLOPs. For large n the AP1000 consistently achieves in the range 79 to 82 percent of its theoretical peak (with the ratio slightly better when the number of cells is a perfect square, e.g. 64 or 256, than when it is not).

An encouraging aspect of the results is that the AP1000 has relatively low n_{half}. For example, on the 64-cell AP1000 at ANU we obtain at least half the maximum performance (i.e. at least 145 MFLOPs) for problem sizes in the wide range $648 \leq n \leq 10000$. (On the 64-cell Intel Delta, the corresponding range is $2500 \leq n \leq 8000$ [10].) As expected from (5.6), n_{half} is roughly proportional to $P^{1/2}$.

cells	r_{max} GFLOPs	n_{max} order	n_{half} order	r_{peak} GFLOPs	r_{max}/r_{peak}
512	2.251	25600	2500	2.844	0.79
256	1.162	18000	1600	1.422	0.82
128	0.566	12800	1100	0.711	0.80
64	0.291	10000	648	0.356	0.82
32	0.143	7000	520	0.178	0.80

Table 5: LINPACK Benchmark results for large n

Because of the influence of the cache and the effect of blocking, the formula (5.5) gives a good fit to the benchmark results only if n is sufficiently small and ω is fixed (or blocking is not used).

5.3 Optimizations for BLAS-3 Triangular Matrix Updates

If B is an $m \times n$ matrix, to form $B \leftarrow A^{-1}B$, where A is an $m \times m$ upper triangular matrix with unit diagonal, we can perform the corresponding (parallel) rank-1 updates:

$$B \leftarrow B - \tilde{A}_{.j} B_{j.}, \quad \text{for } j = m-1, ..., 1$$

where $\tilde{A} = A - I$. A straightforward ('unblocked') implementation on the AP1000 uses row/column broadcasts and rank-1 updates. However, performance can be improved by grouping w updates together, as described in Section 5.1.

Table 6 gives results for this computation for single precision, with $\omega = 4N_y\sqrt{n/(2N_y)}$. For the unblocked algorithm, the performance does not even approach that of `Rank1Update()`, due to communication overheads (for small n) and the fact that rank-1 update is a Level 2 operation and hence makes poor use of the cache (for large n). For the blocked algorithm, performance is better but still does not approach that of `UpdateRect()`, due to the fact that the optimal ω is a tradeoff between seeking a higher proportion of the computation in `UpdateRect()` (needing a low ω) and seeking a high number of iterations in each call to `UpdateRect()` (needing a high ω).

A value of $\omega \simeq k_0$ (Section 4.2) is optimal for `UpdateRect()`. The tradeoff mentioned above can be overcome by recursively applying the blocking process described in Section 5.1, for $\omega \simeq k_0, k_0/2, k_0/4$, etc. As larger values of ω are now used, the partitioning methods of Section 4.2 must also be employed. The performance for moderate sized matrices of this 'super-blocked' scheme is given in Table 6; for larger matrices, performance steadily improves up to 7.3 MFLOPs for $n/N_x = 1024$. These results indicate that the AP1000 can perform BLAS-3 triangular matrix updates at 85% of the its theoretical peak speed.

While the coding of such a recursive blocking scheme is complex, it could be similarly applied to the more complex LINPACK benchmark, with similar improvements in performance to be expected.

6 Conclusions

In this paper, we have described implementations of the BLAS-3 and the LINPACK Benchmark on the Fujitsu AP1000. Many of the techniques presented, such as the design of SPARC BLAS-2 and BLAS-3 kernels (Section 3), partitioning methods for direct-mapped caches (Section 4.2),

n/N_x	unblocked			blocked			super-blocked		
	N_x=1	2	8	N_x=1	2	8	N_x=1	2	8
32	4.6	3.8	3.8	4.0	3.8	4.0	4.2	3.6	3.7
64	5.4	5.0	5.0	5.6	5.6	5.6	5.8	5.3	5.5
128	5.2	5.1	5.1	6.3	6.4	6.4	6.6	6.4	6.5
180	5.5	5.4	5.4	6.5	6.4	6.6	6.8	6.7	6.8
256	4.3	4.2	4.3	6.7	6.8	6.2	6.8	6.8	6.9
360	3.8	3.7	3.7	6.8	6.9	6.5	7.1	7.1	7.2

Table 6: Speed in MFLOPs/cell for $B \leftarrow A^{-1}B$ for $n \times n$ matrices on the AP1000 (single precision)

and blocking (Sections 5.1 and 5.3) are also applicable to the implementation of other linear algebra applications, on the AP1000 and on similar architectures.

The LINPACK Benchmark and BLAS-3 results show that the AP1000 is a good machine for numerical linear algebra, and that on moderate to large problems we can consistently achieve close to 80% of its theoretical peak performance, for the former, and 85-90% for the latter. They signify that the AP1000 architecture is well balanced on all levels, with respect to floating point computation. The main reason for this is the high ratio of communication speed to floating-point speed compared to machines such as the Intel Delta and nCUBE. The high-bandwidth hardware row/column broadcast capability of the AP1000, extremely useful in linear algebra applications, and the low latency of the send/receive routines are also significant. As shown in Table 1, the speed of the former make the use of 'systolic' versions of linear algebra algorithms unnecessary. The large, direct-mapped cache, while requiring extra effort for full optimization, and the large cell memory are also very important features.

References

[1] R. P. Brent, "The LINPACK Benchmark on the Fujitsu AP 1000", *Frontiers '92*, Virginia, USA, , October 1992.

[2] R. P. Brent (editor), *Proceedings of the CAP Workshop '91*, Australian National University, Canberra, Australia November 19901.

[3] R. P. Brent and M. Ishii (editors), *Proceedings of the First CAP Workshop*, Kawasaki, Japan, November 1990.

[4] 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.

[5] 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.

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

[7] J. J. Dongarra, *Performance of various computers using standard linear equations software*, Report CS-89-05, Computer Science Department, University of Tennessee, version of November 12, 1991

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

[9] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon and D. W. Walker, *Solving Problems on Concurrent Processors, Vol. I: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[10] R. A. van de Geijn, *Massively parallel LINPACK benchmark on the Intel Touchstone DELTA and iPSC/860 systems*, Report TR-91-92, Department of Computer Sciences, University of Texas, August 1991.

[11] G. H. Golub and C. Van Loan, *Matrix Computations*, Johns Hopkins Press, Baltimore, Maryland, 1983.

[12] G. Li and T. F. Coleman, "A new method for solving triangular systems on distributed-memory message-passing multiprocessors", *SIAM J. Sci. and Statist. Computing* 10 (1989), 382-396.

[13] G. W. Stewart, *Introduction to Matrix Computations*, Academic Press, New York, 1973.

[14] P. E. Strazdins and R. P. Brent, "Implementing BLAS level 3 on the CAP-II", in [3].

[15] P. E. Strazdins and R. P. Brent, "Implementing BLAS level 3 on the AP1000", in [2].

[16] Sun Microsystems, *Sun-4 Assembly Language Reference Manual*, May 1988.