References

[1] Carter, L., Alpern, B., *Private Communication*

[2] Demmel,J., W., Dongarra, J. J., Du Croz, J., Greenbaum, A., Hammarling, S., and Sorensen, D. C. 1987 "Prospectus for the development of a linear algebra library for high-performance computers. "

[3] Dongarra, J. J., Du Croz, J., Hammarling, and Hanson, R. J. 1988a "An extended set of Fortran basic linear algebra subprograms." *ACM Trans. Math. Software 14:1-17;18-32*

[4] Dongarra, J. J., Du Croz, J., Hammarling, and Duff, I. S., 1988b "A set of Level 3 basic linear algebra subprograms," *Report AERE R 13297. Oxford: Computer Science and Systems Division, Harwell Laboratory.*

[5] Dongarra, J.J. , Mayes, P., di Brozolo, G.R. "The IBM RISC System/6000 and Linear Algebra Operations" *University of Tennessee Computer Science Tech Report: CS - 90 - 122*

[6] Gallivan, K., Jalby, W., Meier, U., Sameh, A., " Impact of hierarchical memory systems on linear algebra algorithm design. " *The International Journal of Supercomputer Applications, V. 2, No. 1, Spring 1988, pp. 12-48.*

[7] Golub, G.H., and Van Loan, C., Matrix Computations, 2nd ed.. *The Johns Hopkins University Press, 1989*

[8] Gustavson, F., *Private Communication*

[9] Kågström, B., and Van Loan, C., "GEMM-Based Level-3 BLAS." *Theory Center Technical Report,* January 1991.

[10] Lawson, C. L., Hanson, R. J., Kincaid, D. R., and Krogh, F. T. 1979a. "Basic linear algebra subprograms for Fortran usage." *ACM Trans. Math. Software 5:308-323.*

[11] — 1979b. "Algorithm 539. Basic linear algebra subprograms for Fortran usage. " *ACM Trans. Math. Software 5:324-325.*

[12] IBM, Engineering and Scientific Subroutine Library, "Guide and Reference", Release 4

[13] Van Loan, C. Matrix Frameworks for Fast Fourier Transform *SIAM Publications, Philadelphia*, 1991

7 Conclusions

A brief survey of the results indicates that applying a block data structure gives a 5 to 10 percent improvement on rectangular DGEMMs and a marginal improvement on square DGEMMs. More importantly, notice that in the rectangular case, the block structures hit the reasonable speed of 45 Megaflops when K was around 60 to 70. The other code did not even hit that speed until K was way past 120 (see also table 2). We encourage people to try these ideas for themselves, since there is some question as to how fast non-block data structured DGEMMs can run. This indicates that there would be considerably less level 2 computation in a major algorithm like Cholesky or LU.

By improving the speed of the DGEMM and forcing its peak performance sooner, this is a double bonus enabling better performanced on any of the BLAS or other routines that use DGEMM as a kernel.

8 Acknowledgements

This research was partially supported by the Cornell Center for Theory and Simulation in Science and Engineering, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute. The research was conducted using an IBM RISC System/6000s Model 530. Many of the ideas in this paper were discussed with L. Carter and B. Alpern of IBM Research, Yorktown Heights. We thank A. Hoisie of the Cornell Theory Center for some useful ideas. We also thank F. Gustavson of IBM Research, for his implementations of some of these ideas as well as suggestions on an early pre-print. Finally, we thank C. Van Loan, for his suggestions during all levels of the previous drafts.

touches a new column. This is especially suitable for BLAS that have small blocks. This is not to say that the block data structure is slower in other cases, but if, for example, the blocks are at most a cache line size (16 for RS/6000 Models 530 and above) by the number of cache associativity classes (4 for the RS/6000), then the block data structure is works at optimal efficiency and is marginally faster then the regular transposition.

The algorithm used was quite simple.

```
**Alg.1:** Copy $(A^{P(m,n)})_{MN\times 1} \leftarrow A^T$
    $index = 1$
    for $J = 1, K$
        for $I = 1, R$
            for $JJ = 1, m$
                for $II = 1, n$
                    $A^P(index) = A^T(II, JJ)$
                    $index = index + 1$
            end for $JJ$
        end for $I$
    end for $J$
```

```
**Alg.2:** Copy $(A^{P(m,n)})_{MN\times 1} \leftarrow A$
    $index = 1$
    for $Rblock = 1, R, Someblock$
        for $J = 1, K$
            for $I = Rblock, Rblock + Someblock$
                for $II = 1, n$
                    for $JJ = 1, m$
                        $A^P(index) = A(II, JJ)$
                        $index = index + 1$
                    end for $JJ$
                end for $II$
            end for $I$
        end for $J$
    end for $Rblock$
```

Order	*TransA*	*TransB*	*Alg. I*	*Alg. II*	*Alg. IV*	*Instances*
100	'N'	'N'	43.05	41.14	41.27	259
100	'N'	'T'	n/a	41.62	39.83	259
100	'T'	'N'	43.52	41.44	41.7	259
100	'T'	'T'	n/a	41.92	37.13	259
200	'N'	'N'	44.73	44.09	43.52	26
200	'N'	'T'	n/a	44.29	42.91	26
200	'T'	'N'	44.78	44.14	43.9	26
200	'T'	'T'	n/a	44.24	43.28	26
300	'N'	'N'	44.95	44.58	44.18	8
300	'N'	'T'	n/a	44.79	43.83	8
300	'T'	'N'	45.1	44.69	44.18	8
300	'T'	'T'	n/a	45.0	44.38	8
400	'N'	'N'	45.22	45.24	44.52	3
400	'N'	'T'	n/a	45.24	43.79	3
400	'T'	'N'	45.27	45.22	44.78	3
400	'T'	'T'	n/a	45.22	44.2	3
500	'N'	'N'	45.34	45.48	44.65	1
500	'N'	'T'	n/a	45.3	44.06	1
500	'T'	'N'	45.34	45.34	44.91	1
500	'T'	'T'	n/a	45.26	44.48	1
600	'N'	'N'	45.38	45.43	45.0	1
600	'N'	'T'	n/a	45.53	44.46	1
600	'T'	'N'	45.48	45.63	44.89	1
600	'T'	'T'	n/a	45.53	45.04	1

Table 2: Order of Square DGEMMS vs. Speed

Order	*Algorithm I*	*Algorithm IV*
300	34.0	32.7
400	35.6	33.9
500	37.3	36.1

Table 3: QR Factorization Based on Different DGEMM, NB=21

K	$TransA$	$TransB$	I	II	III	IV	V	$Instances$
30	'N'	'N'	42.94	42.14	n/a	37.24	36.74	23
30	'N'	'T'	n/a	42.03	n/a	37.63	33.05	23
30	'T'	'N'	43.05	42.06	35.8	37.32	36.61	23
30	'T'	'T'	n/a	42.03	n/a	37.66	35.55	23
40	'N'	'N'	44.0	43.48	n/a	39.16	39.23	17
40	'N'	'T'	n/a	43.32	n/a	39.84	35.24	17
40	'T'	'N'	43.95	43.4	37.86	39.23	39.16	17
40	'T'	'T'	n/a	43.32	n/a	39.58	38.03	17
50	'N'	'N'	44.58	44.22	n/a	40.48	40.54	14
50	'N'	'T'	n/a	44.1	n/a	40.81	36.54	14
50	'T'	'N'	44.66	44.0	39.07	40.38	40.54	14
50	'T'	'T'	n/a	44.03	n/a	40.71	39.64	14
60	'N'	'N'	45.05	44.57	n/a	41.6	41.56	11
60	'N'	'T'	n/a	44.92	n/a	41.45	36.04	11
60	'T'	'N'	45.18	44.74	39.9	41.41	41.74	11
60	'T'	'T'	n/a	44.57	n/a	41.78	40.76	11
70	'N'	'N'	45.35	45.06	n/a	41.82	38.86	10
70	'N'	'T'	n/a	45.14	n/a	42.17	33.81	10
70	'T'	'N'	45.43	45.10	40.26	41.82	38.92	10
70	'T'	'T'	n/a	45.18	n/a	42.32	37.64	10
80	'N'	'N'	45.66	45.45	n/a	42.2	39.89	8
80	'N'	'T'	n/a	45.23	n/a	42.36	34.75	8
80	'T'	'N'	45.5	45.23	40.49	42.4	39.89	8
80	'T'	'T'	n/a	45.15	n/a	42.38	38.66	8
100	'N'	'N'	45.74	45.4	n/a	42.95	41.22	7
100	'N'	'T'	n/a	45.7	n/a	43.06	36.19	7
100	'T'	'N'	45.74	45.83	41.53	43.1	41.29	7
100	'T'	'T'	n/a	45.91	n/a	43.59	40.28	7
120	'N'	'N'	45.29	45.76	n/a	43.51	42.16	5
120	'N'	'T'	n/a	45.66	n/a	43.31	36.21	5
120	'T'	'N'	45.37	45.56	42.03	43.42	42.33	5
120	'T'	'T'	n/a	45.76	n/a	43.47	41.29	5

Table 1: K vs. Rectangular DGEMMs (M=585,N=595,Block=39)

blocking parameter varied depending on the input, but it was usually around 100 to 120.

For a final comparison, we ran a QR factorization, which is heavily dependent on GEMMs, to see the improvement possible in that case. On a matrix of order 500, the fastest QR performance we saw was around 36.1 Mflops on the 530. No special optimization techniques were used besides employing a standard block algorithm. If we used an alternate GEMM for some of the larger multiplies, the same algorithm improved to 37.3 Mflops, which was not as impressive as we had hoped. The final table summarizes some of the results.

5 Non-BLAS Block Data Structures

It is reasonable that higher level computation could benefit from block data structures. Since an alternate data structure-based code is most useful in a kernel for other code, the most obvious application is in the BLAS. This is especially true since one may only need to write the code to do GEMM work and then use this to write the other BLAS with the only concern being blocking sizes.

6 The Actual Algorithm of Block Transposition

The remaining theoretical task is to justify that this alternate block data structure for A^T is a time saver. As is noted, using a matrix already in this form might be faster, but since this is a nonstandard data structure, we must assume that we need to copy it into this form. By earlier assumption, the cost of copying into A^T must be small enough for any of this to even be considered. It is now noted that for certain blocking sizes, the copying into A^P requires *less* time then the copying into A^T. Since this is not a paper on transposition only a cursory justification in our example will be given.

Comparing our "unrealistic" example given in the introduction of the paper, suppose one is storing and computing A^T or A^P sequentially. The first place they differ is in the (31) position ($A^T(3,1) = a_{13}$ and $A^P(3,1) = a_{21}$). The transpose is accessing something further away in memory since it is accessing across a row. The block transpose does not go as far out in a row and accesses data down a column once it

DGEMM.

- Algorithm III: A Blocked, nonoptimal, Fortran DGEMM. (Alg.2 in the text performed slightly better, usually somewhere between the next two algorithms. What we did here was provide the exact same blocking as done in our block data structure, even though that might not have been optimal for the standard data structure. This is basically shown as a low end result.).
- Algorithm IV: The ESSL release of DGEMM. We note that the ESSL timings for Algorithm IV were higher than the results we obtained. We are trying to establish a reason for this discrepancy. Part of this reason may be the fact that we used repeated instances as described above or perhaps the usage of a different timer or compiler version.
- Algorithm V: This is the DGEMM reported in [5]. Since this code was obtained later, these timers had to be done separately, but the conditions of the test remained the same. For example, the $LDA's$ in the test cases were always fixed at 600.

For Algorithms I and II the time for copying into the data structure is included in the timing results. To ignore this copying time in talking about the results would be a serious oversight. So the timing reported is for calling the copy routines and then calling the algorithms. Otherwise, the results would look unrealistically better.

In Table I, we compare for approximately $M = 600, N = 600$, different algorithms versus different values of K. For larger K, Algorithms I and II are further blocked along K. The numbers under each Algorithm was the reported Megaflop rate for the RS/6000 Model 530 (whose theoretical peak performance is 50 Megaflops). The program timing these results used all of the algorithms, with the same matrix, at the same time, to ensure fairness. The only exception to this was a separate code used for the blocked versions of algorithms I and II in Table II.

In Table II, we compare almost square DGEMMs of various orders. Here is where the more typical algorithms will get peak performance. Algorithms I and II were further blocked in the K direction. The

Most algorithms for DGEMM hit peak speed when the matrix dimensions, M, N, K, are comparatively big (small enough to fit into memory, but larger than size of the memory structure beneath the main memory). Unfortunately, in algorithms like Cholesky, LU, and QR, DGEMM calls invariably are rectangular with one dimension larger than the other. We feel that in the event of block algorithms, most cases of GEMM calls have at least one of the dimensions much smaller than the others. We feel it necessary to consider the case where the K dimension is small (10 to 120) for this reason. Typically, we would like the DGEMM to run with peak speed at the smallest possible K we can afford. There are two reasons for this. The first is that a small K represents less Level-2 work, which is slower. The second is that if we can get a faster algorithm, clearly the problem will be solved faster.

When the alternate data structures to A^T previously were introduced, a very specific data structure was introduced along with them. In afterthought, this data structure is designed to exploit a faster peek speed in the cases when K is small. If K is larger, then the algorithm must be blocked once more on the outside with respect to K.

Finally, buffers needed to be around large enough to do the transpositions. In a sense, this is cheating and so of course our results are better. In another sense, the purpose of this paper was to determine if such a method could be useful. Indeed, we shall see it is.

4 Numerical Results

For each timing, we ran enough examples so that DGEMM would run for at least 10 system seconds. We include in our tables exactly how many instances that took. Sometimes more than one run of 10 or more seconds each was made, to help balance out system loads and the possible inaccuracy of the timer. In these cases, the times between different runs were averaged.

There are five different algorithms used.

- Algorithm I: The Block Data Structure DGEMM mentioned in the text.
- Algorithm II: The so-called Twice done Block Data Structure

3.5 Twice Done Block Data Structure for the Transpose

One might wonder if we could extend this idea until we find $A^P B^P$. We call this a *twice done block data structure*. The integer arithmetic increases again.

```
Alg.4: Find $C \leftarrow C + A^P B^P$
    $nb = 0$
    for $N2 = 1, N, Block$
        $ma = 0$
        $nbsave = nb$
        for $M1 = 1, M, m = 7$
            $ma = ma + 1$
            $nb = nbsave$
            for $N1 = N2, N2 + Block - n, n = 3$
                $nb = nb + 1$
                $CTEMP = zeros(m - 1, n - 1)$
                $k1 = 1$
                for $KA = 1, m * K, m = 7$
                    $ATEMP = A(ka : ka + m - 1, ma)$
                    $BTEMP = B(K1 : K1 + n - 1, nb)$
                    (Compute $CTEMP = ATEMP^T BTEMP$)
                    $k1 = k1 + 3$
                end for $KA$
                $C(M1 : M1 + m - 1, N1 : N1 + n - 1) =$
                $C(M1 : M1 + m - 1, N1 : N1 + n - 1) + CTEMP$
            end for $N1$
        end for $M1$
    end for $N2$
```

Notice that some changes regarding the variables in the innermost loop are in effect. This is to help offset additional integer arithmetic.

3.6 Important Note on the Use of Block Data Structures

We should note that it was possible to model the transposition time on the RS/6000 Model 530. This in turn made it possible to ascertain in advance when it was worthwhile.

```
    end for $M2$
end for $N2$
```

3.4 Block Data Structured DGEMM

If we look back at the code (Alg.1) to do the transposed DGEMM we notice that on the innermost loop, we access A across rows in 1×7 blocks. One might wonder what would happen if the access of A was sequential even in this innermost loop. The new data structure would still be quite similar to the transpose but it would have rows that were 7 times as long. We consider the new algorithm for the case $m = 7$, $n = 3$.

```
Alg.3: Find $C \leftarrow C + A^P B$
    for $N2 = 1, N, Block$
        $ma = 0$
        for $M1 = 1, M, m = 7$
            $ma = ma + 1$
            for $N1 = N2, N2 + Block - n, n = 3$
                $CTEMP = zeros(m-1, n-1)$
                $ka = 1$
                for $K1 = 1, K$
                    $ATEMP = A(ka : ka + 6, ma)$
                    $BTEMP = B(K1, N1 : N1 + n - 1)$
                    (Compute $CTEMP = ATEMP^T BTEMP$)
                    $ka = ka + 7$
                end for $K1$
                $C(M1 : M1 + m - 1, N1 : N1 + n - 1) =$
                $C(M1 : M1+m-1, N1 : N1+n-1)+CTEMP$
            end for $N1$
        end for $M1$
    end for $N2$
```

The obvious disadvantages to the above mentioned algorithm is that it has an increased amount of integer arithmetic. We hope that the decrease in cache misses will help offset this difficulty.

$$mn + m + n - \lfloor \frac{m-1}{n} \rfloor \leq 32, n \leq m$$

The answer to this is $(m, n) = (6, 4)$. When we compiled our $6x4$ code, however, our version of the fortran compiler was not able to reuse these registers. Fred Gustavson [8] discovered that an older version of the compiler was able to compile the $6x4$ case without spilling the registers. We were using IBM AIX XL Fortran version 02.01.0000.0000, which in spite of dozens attempts, we were unable to get this version to handle $6x4$ case effectively. One should keep in mind that it might be possible to improve our results using an older version of the compiler.

3.3 Small K Fortran DGEMM

Since later in the paper we emphasize the small K dimension sizes, it seems reasonable to note how the algorithm in the previous section changes when K is small. The algorithm is mostly the same, but we have found that blocking along M helps offset the fact that K is small. The innermost blocking is still determined by the information in the previous section. The outermost blocking now has one more additional level of complexity. $Mblock$ ranged from 35 to 63 and $Nblock$ ranged from 117 to 150 during the computation of Table One in the numerical results section.

```
Alg.2: Find $C \leftarrow C + A^T B$ ($K$ small)
for $N2 = 1, N, Nblock$
    for $M2 = 1, M, Mblock$
        for $N1 = N2, N2 + Nblock - n, n$
            for $M1 = M2, M2 + Mblock - m, m$
                $CTEMP = zeros(m-1, n-1)$
                for $K1 = 1, K$
                    $ATEMP = A(K1, M1 : M1 + m - 1)$
                    $BTEMP = B(K1, N1 : N1 + n - 1)$
                    (Compute $CTEMP = ATEMP^T BTEMP$)
                end for $K1$
                $C(M1 : M1 + m - 1, N1 : N1 + n - 1) =$
                $C(M1 : M1 + m - 1, N1 : N1 + n - 1) + CTEMP$
            end for $M1$
        end for $N1$
end for $N2$
```

For this block version of DGEMM, it is necessary to consider blocking for all of the levels of memory. On the innermost level we have registers. We have 32 registers we can use. That means we can allocate at most this many temporary variables. All the C variables take up at least mn registers. The A and B temporary variables take up at most $m+n$ registers. We wish to maximize the amount of multiplies, mn, compared to the amount of loads, $m+n$ while trying to keep $mn+m+n$ manageable (around 32 or less). There is an obvious symmetry between m and n, so we note that since we want to store C as much by column as possible, that $n \leq m$. We now have a simple optimization routine:

$$\max_{(m,n)\text{ integer}} (mn - m - n) \text{ subject to } mn + m + n \leq 32, n \leq m$$

The answer to this problem is $(m,n) = (7,3)$ or $(5,4)$. As mentioned before, we would like n as small as possible, yielding a preference of $(m,n) = (7,3)$. That means in the innermost loop there are 10 loads and 21 multiplies giving a total of 11 free cycles to absorb a cache miss. It is assumed that the loads and multiplies will be arranged a bit more intelligently than written in the above algorithm.

This handles registers and allows cache-misses to be minimalized. The *Block* parameter is designed to pay attention to the TLB. It has been found that for an LDB of around 600 that *Block*s ranging from 40 to 80 is wise.

We note that it is possible on some machines to have register re-use. That is, we do not need $m+n$ registers for the loading of A and B, but $m+n-\lfloor\frac{m-1}{n}\rfloor$ will suffice. This is the approach taken by Bowen Alpern [1] in the assembly DGEMM mentioned. We can do this when it is possible to use all the order the loading of A and B such that we multiply all the n variables loaded from B by the extra $\lfloor\frac{m-1}{n}\rfloor$ variables loaded from A so that the respective registers are available for reuse later.

Note that the new optimization problem becomes

$$\max_{(m,n)\text{ integer}} (mn - m - n) \text{ subject to}$$

The assembly code for DGEMM was thoroughly optimized by Bowen Alpern and Larry Carter at IBM Research Center at Yorktown Heights [1]. The fortran code for DGEMM was written by the author and is described below. Also used for the basis of comparison was the ESSL release of DGEMM [12]. The code for the block data structure DGEMM will also be described below.

3.2 Fortran DGEMM

One fast case of DGEMM is the double precision computation of $C \leftarrow \alpha A^T B + \beta C$. This particular case seems to access memory more consecutively than the other others. Here, we assume C is $M \times N$, A^T is $M \times K$, and B is $K \times N$. We also express the leading dimensions of A, B, and C by LDA, LDB, and LDC respectively. The user supplies A, B, C, α, β, LDA, LDB, and LDC. For the purposes of discussion only, it is convenient to suppress the α and β by assuming they are equal to 1. We also make the assumption that M, N, K are divisible by whatever small numbers we choose for convenience of coding. This last assumption is certainly not necessary, but allows for the discussion of the computation of DGEMM on all but at most a few rows and columns which can be handled separately in lower order work.

We go immediately into a block version of DGEMM, discussing how to find the optimal blocking parameters afterwards.

```
Alg.1: Find $C \leftarrow C + A^T B$
        for $N2 = 1, N, Block$
            for $M1 = 1, M, m$
                for $N1 = N2, N2 + Block - n, n$
                    $CTEMP = zeros(m-1, n-1)$
                    for $K1 = 1, K$
                        $ATEMP = A(K1, M1 : M1 + m - 1)$
                        $BTEMP = B(K1, N1 : N1 + n - 1)$
                        (Compute $CTEMP = ATEMP^T BTEMP$)
                    end for $K1$
                    $C(M1 : M1 + m - 1, N1 : N1 + n - 1) =$
                    $C(M1 : M1 + m - 1, N1 : N1 + n - 1) + CTEMP$
                end for $N1$
            end for $M1$
```

Then our example of an alternate data structure for A^T is

$$A^P = \left[\begin{array}{c|c|c|c} a_{11} & a_{13} & a_{31} & a_{33} \\ a_{12} & a_{14} & a_{32} & a_{34} \\ a_{21} & a_{23} & a_{41} & a_{43} \\ a_{22} & a_{24} & a_{42} & a_{44} \end{array}\right]$$

It should be noted that the block data structure stores its data as follows:

$$A^P \leftarrow \left[\begin{array}{c} A_{11} \\ A_{21} \\ A_{12} \\ A_{22} \end{array}\right]$$

where A_{ij} was defined above and each A_{ij} started a new column in A^P. In less degenerate cases, several block matrices will be required before a new column is necessary (as an aside, one could just use a vector for A^P if it weren't for the fact that the indexing overhead of the last entries in a large matrix would become hard to manage).

It should be clear that an algorithm could only benefit from knowing it could access data sequentially. Of course, one needs to decide on blocking sizes before running a transpose algorithm. Not only does this necessitate more than one transpose algorithm for a set of BLAS, but it also requires routines to be written for a whole new data structure as well.

3 Using A Block Data Structure for DGEMM

3.1 IBM Superscalar RISC S/6000

Since the BLAS themselves are machine specific, it should come as no surprise that any algorithm we present for DGEMM must likewise be machine specific. The architecture we tested this on is the IBM RISC System/6000 Model 530 and Model 520 series. This is a superscalar machine which pipelines up to five instructions per cycle. It consists of four levels of hierarchical memory: main memory, a TLB (translation lookahead buffer), cache, and registers. Data movement between the levels varies from 8 to 34 cycles, so it is to the users advantage to block for each level of hierarchy to minimize data loading overhead.

$$y = \Pi^T_{n_1,n_2} x = \left[\begin{array}{c} x(0 : n_1 : n_2) \\ x(1 : n_1 : n_2) \\ \vdots \\ x(n_1 - 1 : n_1 : n_2) \end{array} \right]$$

See Van Loan[13], for example. Then if X is $n_1 \times n_2$ but viewed as a one dimensional array, which we denote $X_{n_1 n_2 \times 1}$,

$$y_{n_2 \times n_1} = \Pi^T_{n_1 \times n_1 n_2} X_{n_1 n_2 \times 1} = (X_{n_1 \times n_2})^T$$

Defining standard transposition in term of a permutation matrix. The alternate to the transpose, $A^P_{(m,n)}$ of A^T is just:

$$A^P_{(m,n)} = (I_{mR} \otimes \Pi^T_{R \times mR}) A^T_{n \times RK}$$

Where I is the square identity of the appropriate dimension, and $\otimes$ represents the Kronecker product. Then the block data structure of A is

$$A^P_{(m,n)} = (I_{mR} \otimes \Pi^T_{R \times mR})((\Pi^T_{M \times MN} A_{MN \times 1})_{n \times RK})$$

To illustrate this block data structure, it is convenient to consider an *unrealistically* small example (4×4). The unrealistic is emphasized to point out that this is given for illustration purposes alone. Here, $M = N = 4$, $m = n = 2$, $R = N = 2$. Both A, A^T, and A^P are given.

$$A = \left[\begin{array}{cc|cc} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{array} \right] A^T = \left[\begin{array}{cc|cc} a_{11} & a_{21} & a_{31} & a_{41} \\ a_{12} & a_{22} & a_{32} & a_{42} \\ \hline a_{13} & a_{23} & a_{33} & a_{43} \\ a_{14} & a_{24} & a_{34} & a_{44} \end{array} \right]$$

If A^T is to be blocked into 2×2 chunks then:

$$A^T = \left[\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right]$$

Where, for example,

$$A_{12} = \left[\begin{array}{cc} a_{31} & a_{41} \\ a_{32} & a_{42} \end{array} \right]$$

the matrix.

There are a few important points to make before going into an example. The first point is that the motivation behind using a block data structure is to achieve faster performance. Not only does it do this, but it achieves the top speeds on smaller problems. The next thing is that we are really solving the same problem but in a block space. One may decide to start off a sequence of BLAS, say at the beginning of a factorization, by placing everything into block data structures and just remaining there. The advantage to this is that it avoids the cost of all the transformations each stage. But another point not to be overlooked is that use of a block data structure makes coding easier. For example, if the user is solving the above DGEMM, with either A^TB or AB, and decides that A will be in an block data structure, then the user no longer needs two separate codes for the A^TB and the AB case. The user just needs one case, and two separate algorithms for converting A^T or A into this case.

2 An Example

To the end of defining our particular block data structure, we write A as $M \times N$ and therefore A^T as $N \times M$. We assume $M = Km$ and $N = Rn$. We also assume K is divisible by n.

$$A^T = \begin{bmatrix} A_{11}^T & A_{12}^T & \cdots & A_{1K}^T \\ A_{21}^T & A_{22}^T & \cdots & A_{2K}^T \\ \vdots & \vdots & \ddots & \vdots \\ A_{R1}^T & A_{R2}^T & \cdots & A_{RK}^T \end{bmatrix}$$

If we consider the standard store by column data structure used in fortran among other languages then it is clear that unless $m = 1$, the $n \times m$ blocks in A^T are noncontiguous. If the data is reorganized so that these blocks are contiguous, then we write this as $A^{P(m,n)}$ (or just A^P if m, n are known) and call this a block data structure (for A^T).

More precisely, define the permutation matrix Π_{n_1,n_2}^T as follows:

1 Introduction

Portability and efficiency have led to block algorithms based on the BLAS[4]. The BLAS routines are then optimized for different machines. It was suggested in [9] that all the level 3 BLAS could be specified in terms of GEMM (GEneral Matrix Multiply). The new goal is to optimize the GEMM and then all the other routines will run close to optimal as well. Because this one routine is now so important, one strategy typically used is to analyze the GEMM on an assembler level to be sure to get as efficient a routine as possible. Instead, we suggest that the user consider using a block data structures which takes into account the machines architecture. Rather than restructing algorithms to take advantage of the cases our kernel GEMM is fast, we suggest restructing the data so that the kernel GEMM would be faster than any non-restructured code. We specifically discuss an implementation on the IBM RISC System/6000.

Since the overhead in transforming the matrix into a block data structure must be taken into consideration, there are only special cases where this might be appropriate for level 2 BLAS. If the matrix is already in a block data structure, then the results of speedup in the level 2 BLAS will be of interest. Unless stated otherwise, all references to data will implicitly be double precision (hence, our example will be DGEMM). We also assume that the matrix is "big" enough in that the time it takes to transform must be understandably small compared to the work time, and that anybody implementing the ideas in this paper would implement a hybrid algorithm which would default to the standard DGEMM in those cases where it is necessary. More details will be provided later.

It is assumed that the user is working on a machine where the order and access of data is crucial to the performance of the code. We consider the cases of machines like the IBM RISC System/6000 with multiple layers of memory, each of which has a penalty for "misses" (accessing something outside the current layer).

In DGEMM, the case $C \leftarrow \beta C + \alpha A^T B$ runs faster than the case of $C \leftarrow \beta C + \alpha AB$ due to memory access of A. We will denote an alternate to the transpose by $A^P B$ where this final product is the same as $A^T B$ or AB but runs faster than both of them. This a store-by-block block matrix structure specificly adapted from the transpose of

BLAS based on Block Data Structures

Greg Henry

Jaunary 23, 1992

Abstract

The optimization of the BLAS is discussed, with examples given for the IBM superscalar RISC S/6000. The approach suggested is to use block data structures based on store-by-block schemes. We give results and analysis of the optimization of DGEMM. We also suggest how these results can be applied to the higher level factorizations and the other BLAS. Results are given to show the advantages of using block data structures.

Keywords: BLAS, DGEMM, Data Structures
AMS(MOS) subject classifications: 65F99, 68P05

This research was partially supported by the Cornell Theory Center, which receives major funding from the National Science Foundation and IBM Corporation, with additional support from New York State and members of its Corporate Research Institute.