

Recursive Blocked Data Formats and BLAS's for Dense Linear Algebra Algorithms

Fred Gustavson¹, André Henriksson², Isak Jonsson², Bo Kågström², and
Per Ling²

¹ IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598,
U.S.A.

`gustav@watson.ibm.com`

² Department of Computing Science and HPC2N, Umeå University, S-901 87 Umeå,
Sweden

`{andropov,isak,bokg,pol}@cs.umu.se`

Abstract. Recursive blocked data formats and recursive blocked BLAS's are introduced and applied to dense linear algebra algorithms that are typified by LAPACK. The new data formats allow for maintaining data locality at every level of the memory hierarchy and hence providing high performance on today's memory tiered processors. This new data format is hybrid. It contains blocking parameters which are chosen so that the associated submatrices of a block-partitioned A fit into level 1 cache. The recursive part of the data format chooses a linear order of the blocks that maintains a two-dimensional data locality of A in a one-dimensional tiered memory structure. We argue that, out of the NB factorial choices of ordering the NB blocks, our recursive ordering leads to one of the best. This is because our algorithms are also recursive and will do their computations on submatrices that follow the new recursive data structure definition. This is in analogy with the well known principle that the data structure should be matched to the algorithm. Performance results in support for our recursive approach are also presented.

1 Introduction

Today block-based algorithms are the standard in state-of-the-art software for dense linear algebra computations [2]. These algorithms perform most of their computations in calls to high-performance level 3 BLAS [3, 10]. The level 3 BLAS obtain most of their performance by data rearrangements and the use of high-performance atomic kernel routines. The current BLAS are designed to exploit the architecture design while maintaining the functionality of the BLAS and thereby guarantee high performance and portability of dense linear algebra codes. The level 3 factorization algorithms typified by LAPACK make repeated calls to the level 3 BLAS with matrix operands equal to submatrices of fixed size. This results in multiple data copying on operands that are related. How can this be challenged? An answer is to change the data structure of the BLAS input matrices to reflect this relationship, thereby removing any data copying from

the BLAS. The obvious choice is to store matrices as blocks, i.e., submatrices of block-partitioned matrices, instead of the standard column-major (Fortran) or row-major (C) orderings.

One way is to store these blocks in block row or block column formats, with the individual blocks stored in the conventional way. This will give an overall performance improvement since the new BLAS codes become simpler and perform no redundant data copying. However, since the data structure and the algorithms will only partially match, these revised codes will not perform at peak performance.

Recursion is a key concept for matching an algorithm and its data structure. A recursive algorithm leads to automatic blocking which is variable and “squarish” [6]. This layered and variable blocking allow for good data locality, which makes it possible to approach peak performance on today’s memory tiered processors. Using the current standard data layouts and applying recursive blocking have led to faster algorithms for the Cholesky, LU and QR factorizations [6, 11, 4]. However, as we will show here even better performance can be obtained when recursive dense linear algebra algorithms are expressed using a recursive data structure and recursive kernels.

Before we go into any further details we outline the rest of the paper. Section 2 introduces the new recursive blocked data formats. We describe two atomic formats for rectangular and triangular matrices, respectively, that are tailored for recursive dense linear algebra algorithms. In Section 3 we introduce the recursive blocked BLAS’s that support the recursive data formats. We use the packed Cholesky factorization of a positive definite matrix A as a case study. Section 4 describes our design principles for a recursive level 3 algorithm for the general matrix multiply and add (GEMM) operation resulting in a recursion tree with kernel routine calls at the leaves. Finally, in Section 5 we present some performance results from our ongoing developments that strongly support our new recursive approach.

2 Recursive Blocked Data Formats

We introduce a new set of data formats for storing block-partitioned dense matrices. The set is a hybrid of two “addressing” techniques. At the block level each submatrix is stored in the standard column-major (or row-major) order and the size of each block is constrained so that a few of them will simultaneously fit in level 1 cache. Blocks stored in this fashion are operands for the kernel routines. Next we describe the layout of the blocks in memory. To allow for efficient utilization of a memory hierarchy, including efficient cache reuse, we propose a novel method for storing blocks of matrices recursively. We describe two recursive matrix formats: the rectangle and the isosceles right triangle. With these data formats in hand we show that the performance of dense linear algebra computations can be improved.

2.1 Rectangular Recursive Data Format

The metrics of a matrix A of size $M \times N$ stored in ordinary column-major order (Fortran format) include M , N and the leading dimension lda . The latter is used for a proper specification of a subarray of A . For block-partitioning of A we also use two parameters specifying the block sizes, mb and nb , where $1 \leq mb \leq M$ and $1 \leq nb \leq N$. A block-partitioned A consists of $NB = m \cdot n$ blocks A_{ij} of size $mb \times nb$, where $m = \lceil M/mb \rceil$ and $n = \lceil N/nb \rceil$. We use the convention that the last block row and/or block column are padded with zero elements when M and/or N are not multiples of mb and nb . However, no computations on these zero elements are performed. Each submatrix A_{ij} is stored in column-major or row-major order. Notice that padding leads to many economies in code production, e.g., no fix-up code are required in the atomic kernels.

Now, these submatrices can be ordered in $NB!$ different ways. We claim that a recursive ordering is an effective ordering when performing linear algebra operations on the matrix. In fact, we claim that it is better than block row or block column ordering of the blocks. The reason for this is that a recursive blocked format allow for maintaining the two-dimensional data locality at every level of the one-dimensional tiered memory structure, while the block row and block column orderings only maintain data locality at a submatrix level. Irrespective of how the block sizes are set, the block row or block column format can only automatically match one level of the memory hierarchy, e.g., level 1 cache.

The recursive block ordering is determined by always dividing the rectangular dimension of the submatrix that is the largest. The choice when there is a tie leads to two formats: *recursive block row* (RBR): always divide the row dimension; and *recursive block column* (RBC): always divide the column dimension. When dividing an odd number of rows the middle row is assigned to the block at the bottom. For an odd number of columns, the middle one is assigned to the block to the right. The reason is that the block to the right or at the bottom may contain submatrices that are not entirely filled, so this strategy keeps the difference in number of used elements between the two blocks after the splitting to a minimum.

For illustration we consider A of size 496×380 and $mb = nb = 100$, giving $m = 5$ and $n = 4$. Using a block column format, the blocks are mapped as in (1) of Figure 1. The numbers 0–19 denote contiguous blocks in memory. Notice that not all elements in submatrices 4, 9, 14–19 match elements in A , i.e., these submatrices are padded with zero elements.

Using the RBR format for assigning contiguous blocks in memory, we first observe that $5 = m > n = 4$, so the splitting is below the second row ($\lfloor m/2 \rfloor = 2$), assigning the block numbers 0–7 to the upper part and 8–19 to the lower. Since the number of columns (4) is greater than the number of rows (2) in the upper part, the next splitting is vertical, assigning block numbers 0–3 to the left hand side part. This submatrix is now square so the row dimension is split. The complete block assignment associated with the RBR format is displayed in (2) of Figure 1. The procedure is similar when applying the RBC assignment of contiguous blocks, except when splitting square submatrices, see (3). We have

$$\begin{bmatrix} A_{11} & A_{12} & \cdots & A_{14} \\ A_{21} & \ddots & & A_{24} \\ \vdots & & & \vdots \\ A_{51} & \cdots & & A_{54} \end{bmatrix} \sim \begin{pmatrix} 0 & 5 & 10 & 15 \\ 1 & 6 & 11 & 16 \\ 2 & 7 & 12 & 17 \\ 3 & 8 & 13 & 18 \\ 4 & 9 & 14 & 19 \end{pmatrix}_{(BC)} \quad (1)$$

$$\sim \begin{pmatrix} 0 & 1 & | & 4 & 5 \\ 2 & 3 & | & 6 & 7 \\ \hline 8 & 9 & | & 14 & 15 \\ 10 & 11 & | & 16 & 17 \\ 12 & 13 & | & 18 & 19 \end{pmatrix}_{(RBR)} \quad (2)$$

$$\sim \begin{pmatrix} 0 & 2 & | & 4 & 6 \\ 1 & 3 & | & 5 & 7 \\ \hline 8 & 9 & | & 14 & 15 \\ 10 & 12 & | & 16 & 18 \\ 11 & 13 & | & 17 & 19 \end{pmatrix}_{(RBC)} \quad (3)$$

Fig. 1. The mapping of a 5×4 block matrix in block column order (BC), recursive block row order (RBR), and recursive block column order (RBC).

also marked the result of the first recursive splittings for the RBR and RBC formats, see (2) and (3).

The choice of mb and nb is crucial and closely linked to the memory hierarchy of the target architecture. In the extreme case of $mb = M, nb = N$, the entire matrix is stored in one block, which corresponds to the conventional column-major and row-major orderings with their deficiencies like bad data locality in either the row or the column dimension. If $mb = nb = 1$, we apply recursive splitting down to the single elements. This results in good data locality, but the submatrix operations become very inefficient since the kernels will only operate on single elements. For example, register blocking is prohibited. The best choice of mb and nb depends on the size of the level 1 cache. One should strive for fitting one or a few submatrices in level 1 cache. The consequence of having bad data locality in one dimension of the submatrix does not hinder performance, since the level 1 cache is random access. Thus, we have linear addressing in the kernels, which simplifies loop unrolling, preloading and register blocking. The time to calculate the address of an element is $O(\log M + \log N)$. However, this information may be stored in tables, which gives greater flexibility in the placement of the blocks and constant time addressing.

Another benefit of the tables is that the space allocated for each block, S , may be greater than the space required, i.e., the block addresses may be padded so that cache coherency problems may be avoided. For example, the allocation strategy may be to let all blocks begin at a line or page boundary.

2.2 Triangular Recursive Data Format

Since all matrix factorizations can be expressed in terms of rectangular and isosceles triangular matrices it is enough to consider the isosceles case. For an isosceles right triangle of order N , the splitting procedure resembles the rectangular case. Let $nb \times nb$ be the size of the submatrices, giving a block-partitioned triangular A consisting of $n(n+1)/2$ blocks where $n = \lceil N/nb \rceil$. Now divide the triangle into one sub-rectangle and two isosceles sub-triangles. For a lower triangle, the upper left triangle is assigned block numbers 0 to $\lfloor n/2 \rfloor \lfloor (n+1)/2 \rfloor - 1$, the lower right triangle is assigned block numbers $\lfloor n/2 \rfloor \lfloor (n+1)/2 \rfloor$ to $n(n+1)/2 - 1$, and the blocks inside the rectangle will use block numbers $\lfloor n/2 \rfloor \lfloor (n+1)/2 \rfloor$ to $\lfloor n/2 \rfloor \lfloor (n+1)/2 \rfloor + \lfloor n/2 \rfloor \lfloor n/2 \rfloor - 1$. The interior ordering of the blocks in the triangles are determined by applying the algorithm recursively, and the block ordering is either RBR or RBC as in the rectangular case. Figure 2 illustrates the triangular recursive blocked orderings for triangular matrices of order 450, and block size $nb = 100$. The diagonal blocks 0, 2, 9, 12, and 14 are stored in the conventional full format. We use full format since it is easier to write high-performance kernel routines using this format. The blocks in the last block row or block column are possibly padded. The zero blocks in the upper or lower parts, respectively, are not stored, so the space overhead is linear to the matrix order.

$$\begin{aligned}
 \begin{bmatrix} A_{11} & & & & \\ A_{21} & A_{22} & & & \\ \vdots & & \ddots & & \\ A_{51} & \cdots & & A_{55} & \end{bmatrix} &\sim \left(\begin{array}{c|ccc} 0 & & & \\ \hline 1 & 2 & & \\ 3 & 4 & 9 & \\ \hline 5 & 6 & 10 & 12 \\ 7 & 8 & 11 & 13 & 14 \end{array} \right)_{(RBR)} \sim \left(\begin{array}{c|ccc} 0 & & & \\ \hline 1 & 2 & & \\ 3 & 4 & 9 & \\ \hline 5 & 7 & 10 & 12 \\ 6 & 8 & 11 & 13 & 14 \end{array} \right)_{(RBC)} \\
 \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{15} \\ & A_{22} & & \vdots \\ & & \ddots & \\ & & & A_{55} \end{bmatrix} &\sim \left(\begin{array}{c|ccc} 0 & 1 & 3 & 5 & 6 \\ \hline & 2 & 4 & 7 & 8 \\ \hline & & 9 & 10 & 11 \\ & & & 12 & 13 \\ & & & & 14 \end{array} \right)_{(RBR)} \sim \left(\begin{array}{c|ccc} 0 & 1 & 3 & 5 & 7 \\ \hline & 2 & 4 & 6 & 8 \\ \hline & & 9 & 10 & 11 \\ & & & 12 & 13 \\ & & & & 14 \end{array} \right)_{(RBC)}
 \end{aligned}$$

Fig. 2. The RBR and RBC orderings associated with triangular recursive blocked data formats. The splittings from the first recursion step are shown.

From Figure 2 we see that the RBR ordering of a symmetric matrix in lower triangular storage format is equivalent to the RBC ordering of a symmetric matrix in upper triangular storage format, and vice versa.

two branches in the recursion tree that correspond to recursive calls to the Cholesky algorithm with the arguments A_{11} and A_{22} , respectively. In between these calls there are calls to the level 3 BLAS routines for triangular solve of multiple right hands sides (TRSM) and symmetric rank- k update (SYRK). These operations are straightforwardly derived from an appropriate block-partitioning of A and L in $A = LL^T$:

$$\begin{bmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} L_{11} & 0 \\ L_{21} & L_{22} \end{bmatrix} \begin{bmatrix} L_{11}^T & L_{21}^T \\ 0 & L_{22}^T \end{bmatrix} = \begin{bmatrix} L_{11}L_{11}^T & L_{11}L_{21}^T \\ L_{21}L_{11}^T & L_{21}L_{21}^T + L_{22}L_{22}^T \end{bmatrix}. \quad (5)$$

Notice that L overwrites A in algorithm RB-Cholesky. Since the two level 3 operations (TRSM and SYRK) are performed on the A_{ij} blocks in RBR format, we also need recursive algorithms for these operations that match with the blocked recursive data format. We use the convention to add the prefix RB- to the name of a recursive level 3 operation or algorithm. For example, RB-TRSM and RB-SYRK are the two recursive blocked BLAS operations explicitly called in RB-Cholesky.

In this example we have four levels of recursion (0, 1, 2, and 3). Level 3 corresponds to the eight leaves of the binary tree, each calling the Cholesky kernel to factorize one (updated) diagonal block of A in RBR format (labeled 0, 2, 7, 9, 26, 28, 33, and 35). All other nodes perform RB-TRSM and RB-SYRK operations. The four nodes on level 2 work on

$$\left[\begin{array}{c|c} 0 & \\ \hline 1 & 2 \end{array} \right], \quad \left[\begin{array}{c|c} 7 & \\ \hline 8 & 9 \end{array} \right], \quad \left[\begin{array}{c|c} 26 & \\ \hline 27 & 28 \end{array} \right], \quad \left[\begin{array}{c|c} 33 & \\ \hline 34 & 35 \end{array} \right]. \quad (6)$$

Similarly, the two nodes on level 1 work on

$$\left[\begin{array}{c|c|c} 0 & & \\ \hline 1 & 2 & \\ \hline 3 & 4 & 7 \\ \hline 5 & 6 & 8 & 9 \end{array} \right], \quad \left[\begin{array}{c|c|c} 26 & & \\ \hline 27 & 28 & \\ \hline 29 & 30 & 33 \\ \hline 31 & 32 & 34 & 35 \end{array} \right]. \quad (7)$$

Finally, on level 0 there is the root of the tree which is working on the A_{ij} blocks in Figure 3.

The recursive blocked algorithms for the RB-TRSM and RB-SYRK use the GEMM-based approach [9, 5, 10] and therefore call RB-GEMM, which in turn perform all computations at the leaves of its recursion tree by calling register-based GEMM kernel routines with operands that fit in level 1 cache. Our recursive algorithm for RB-GEMM is further discussed in the next section.

4 Recursive Blocked DGEMM

The recursive blocked DGEMM, RB-GEMM, uses a similar divide and conquer algorithm as the RB-Cholesky algorithm (see Figure 6). However, since there are more problem dimensions than for the Cholesky case (one, n , for Cholesky as compared to three, m , n , and k for GEMM), the splitting of the nodes can

take place in one of the three dimensions (see Figure 5). By always splitting the largest dimension, the problem is kept “squarish”, i.e., the ratio between the number of operations made on subblocks and the number of subblocks is maintained as high as possible. Nevertheless, the “conquer” part in GEMM is trivial, since the addition of the results is made implicitly by the leaf kernels. The leaf kernels used are high-performance, prefetching unrolled kernels which utilize the linear addressing with fixed leading dimensions inherited from the block formats. The fixed leading dimensions guarantee that there is no or only small performance differences between different transpose arguments of the GEMM operation ($AB, A^T B, AB^T, A^T B^T$). For a thorough explanation of these kernels, see [7] and [8]. In our view the input matrices which an application (algorithm) will use define how their submatrices will be loaded. Knowing the full algorithm and the machine architecture allow us to choose nb so that misalignment never or rarely occur.

$$\begin{aligned}
& \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \\
& = \begin{bmatrix} [C_{11} \ C_{12}] + [A_{11} \ A_{12}] \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \\ [C_{21} \ C_{22}] + [A_{21} \ A_{22}] \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \end{bmatrix} = \\
& = \left[\begin{bmatrix} C_{11} \\ C_{21} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} \\ B_{21} \end{bmatrix}, \begin{bmatrix} C_{12} \\ C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{12} \\ B_{22} \end{bmatrix} \right] = \\
& = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} + \begin{bmatrix} A_{11} \\ A_{21} \end{bmatrix} [B_{11} \ B_{12}] + \begin{bmatrix} A_{12} \\ A_{22} \end{bmatrix} [B_{21} \ B_{22}]
\end{aligned}$$

Fig. 5. Splitting the matrix multiplication on the m , n , and k dimensions, respectively.

Notice that if $m = n = k > 1$, or two of them are equal, there is a choice on which dimension to split. Some experimental results have shown that this choice is not so crucial. This is due to the fact that a less than optimal ordering of the splittings will only impair performance in one or two levels of recursion. Further down the recursion tree, the impact of previous choices vanishes. In our prototype implementations, we have focused on square subblocks and mostly square matrices. When multiplying submatrices that cross block boundaries, the case of misalignment with the data structure and the algorithm will arise. This is a serious threat to good performance. One way to rectify this problem is to make nb an even power of two, and adjust the algorithm thereafter. This might not handle all cases of misalignment, but should treat the common linear algebra operations.

Implementations have been done in Fortran 77 and in C. The implementation in C is somewhat more readable, since the C language supports recursion, but

by doing clear notations of the explicit stack, the Fortran 77 code is not hard to read.

Level 3 algorithmic prefetching [7, 1] is more difficult to apply in the recursive algorithm, since it is a non-trivial task to calculate the next set of kernel blocks to be multiplied given the current set of blocks. We have looked at three ways to solve this problem. One way is based on the divide-at-even-power-of-two scheme. If we have fixed division points, it is easier to calculate the next set of blocks by using bit arithmetics. Another way of doing prefetching is simply to also include prefetching information (next set of blocks) in the argument list to the RB-GEMM subroutine. A third way of doing prefetching is to in advance build a batch list of the multiplications to be performed. Using this list, finding the next set is trivial. The last method is used in our C implementation [8].

```

function [C(1 : M, 1 : N)] = RB-GEMM (C(1 : M, 1 : N), A(1 : M, 1 : K),
                                         B(1 : K, 1 : N), m, n, k)
if m = 1 and n = 1 and k = 1 then    {Leaf node}
    DGEMM(C(1 : M, 1 : N), A(1 : M, 1 : K), B(1 : K, 1 : N));    {Call kernel routine }
else if m = max{m, n, k} then
    m1 = m/2;  m2 = m - m1;    {# blocks in next recursive calls}
    RB-GEMM (C(1 : m1 · mb, 1 : N), A(1 : m1 · mb, 1 : K),
              B(1 : K, 1 : N), m1, n, k)    {mb is the size of submatrix blocks }
    RB-GEMM (C(m1 · mb + 1 : M, 1 : N), A(m1 · mb + 1 : M, 1 : K),
              B(1 : K, 1 : N), m2, n, k)
else if n = max{n, k} then
    n1 = n/2;  n2 = n - n1;    {# blocks in next recursive calls}
    RB-GEMM (C(1 : M, 1 : n1 · nb), A(1 : M, 1 : K),
              B(1 : K, 1 : n1 · nb), m, n1, k)    {nb is the size of submatrix blocks }
    RB-GEMM (C(1 : M, n1 · nb + 1 : N), A(1 : M, 1 : K),
              B(1 : K, n1 · nb + 1 : N), m, n2, k)
else
    k1 = k/2;  k2 = k - k1;    {# blocks in next recursive calls}
    RB-GEMM (C(1 : M, 1 : N), A(1 : M, 1 : k1 · kb),
              B(1 : k1 · kb, 1 : N), m, n, k1)    {kb is the size of submatrix blocks }
    RB-GEMM (C(1 : M, 1 : N), A(1 : M, k1 · kb + 1 : K),
              B(k1 · kb + 1 : K, 1 : N), m, n, k2)
end

```

Fig. 6. The recursive blocked-GEMM algorithm. This is one of the six (3!) ways of making splitting decisions when all or two of m, n, k are equal to the largest of these dimensions.

Another good characteristic of the recursive algorithm is that it accommodates Strassen's algorithm in a natural way.

5 Performance Results

We present some results from our ongoing software developments. All performance results are obtained on an IBM SMP Node (PowerPC 604, 112 MHz), which is a symmetric shared memory multiprocessing node with four processors. There are two levels of cache memories for each processor. The level 1 cache (L1) is 16 kB on-chip, and the level 2 cache (L2) is a 512 kB off-chip and non-shared memory. The SMP node has a shared 256MB main memory.

In Figure 7 the recursive blocked Cholesky algorithm assuming recursive blocked storage format has the label RBPPF (no copy). The similar algorithm assuming A originally stored in packed lower triangular format is labeled RBPPF (2 copy). In this case, the recursive algorithm is preceded and succeeded by a copy operation between the two data storage formats. Finally, ESSL DPPF denotes the level 3 Cholesky factorization algorithm for packed data from the IBM ESSL library. From these preliminary results we see a great benefit in using the recursive approach (up to 75% and 95%, respectively for large problems using one processor on the IBM SMP Node). There is an additional performance gain when the original data is stored in recursive blocked data format (up to 50% for small problems and around 10% for large problems).

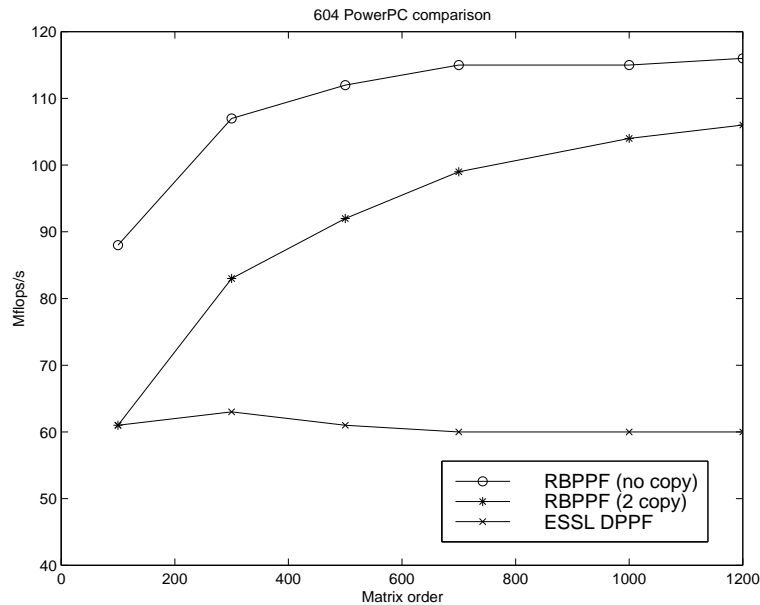


Fig. 7. Performance results for recursive blocked and ESSL packed Cholesky factorization algorithms.

In Figure 8 the recursive GEMM algorithm is compared to a blocked algorithm where the blocks are stored in column block order, i.e., no copying between

conventional column-major order and column block order is included. The column blocked algorithm is optimized for efficient level 1 cache utilization using one level of blocking, 4×4 register blocking and prefetching techniques [8, 7]. The uniprocessor results on the SMP High Node show that the recursive algorithm outperforms the column blocked algorithm for matrices larger than 200×200 . For smaller matrices the overhead for building the recursive tree is showing up. The main reason for the big difference in the performance results for larger problems is that the recursive blocked data format and algorithm provide operands to the kernel routines that are blocked for a memory hierarchy (including level 2 cache and main memory) automatically. In order to get similar performance results for the column blocked algorithm we would have to introduce at least another level of blocking. Such techniques are discussed in [7], where we also present some parallel performance results using a multithreaded recursive GEMM algorithm.

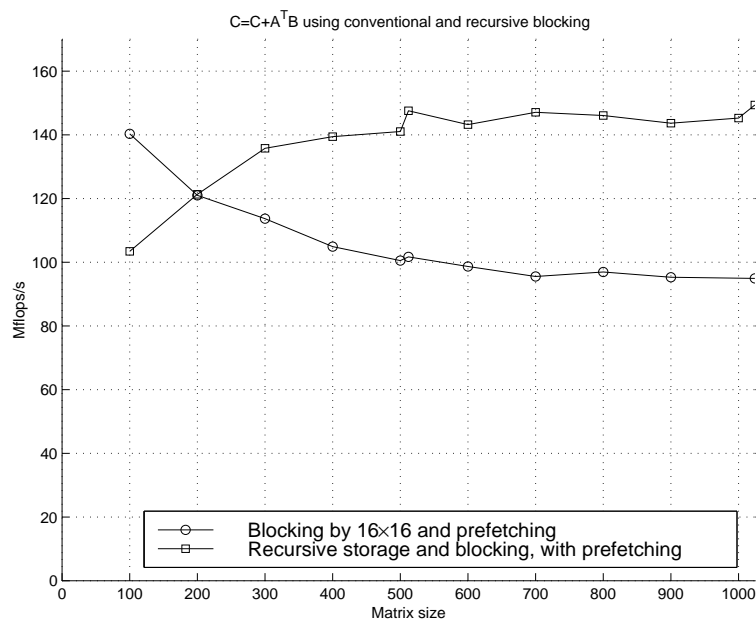


Fig. 8. Performance results for a hand-tuned blocked algorithm using block column format and the recursive GEMM algorithm.

In conclusion, the initial performance results verify the great potential of our recursive approach for the efficient handling of today's memory tiered processors. Our future work will include further developments in support for the recursive blocked data formats and BLAS's.

This research was conducted using the resources of the High Performance Computing Center North (HPC2N).

References

1. R. C. Agarwal, F. G. Gustavson, and M. Zubair. Improving performance of linear algebra algorithms for dense matrices, using algorithmic prefetch. *IBM J. Res. Develop.*, 38(3):265–275, May 1994.
2. E. Anderson, Z. Bai, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen. *LAPACK Users' Guide*, Second Edition. SIAM Publications, Philadelphia, 1995.
3. J. Dongarra, J. DuCroz, I. Duff, and S. Hammarling. A Set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, March 1990.
4. E. Elmroth and F. Gustavson. New Serial and Parallel Recursive *QR* Factorization Algorithms for SMP Systems, *This Proceedings*, Springer Verlag, 1998.
5. IBM. *Engineering and Scientific Subroutine Library, Guide and Reference*, January 1994. SC23–0526–01.
6. F. Gustavson. Recursion leads to automatic variable blocking for dense linear algebra. *IBM J. Res. Develop.*, 41(6):737–755, November 1997.
7. F. Gustavson, A. Henriksson, I. Jonsson, B. Kågström and P. Ling. Superscalar GEMM-based Level 3 BLAS – The Ongoing Evolution of a Portable High-Performance Library. *This Proceedings*, Springer Verlag, 1998.
8. A. Henriksson and I. Jonsson. High-Performance Matrix Multiplication on the IBM SP High Node. *Master Thesis*, UMNAD 98.235, Department of Computing Science, Umeå University, S-901 87 Umeå, June 1998.
9. B. Kågström and C. Van Loan. GEMM-Based Level-3 BLAS. Technical Report CTC91TR47, Department of Computer Science, Cornell University, December 1989.
10. B. Kågström, P. Ling, and C. Van Loan. GEMM-based level 3 BLAS: High-performance model implementations and performance evaluation benchmark. *ACM Trans. Math. Software*, 1997. Accepted for publication.
11. S. Toledo. Locality of Reference in LU Decomposition with Partial Pivoting. *SIAM J. Matrix Anal. Appl.*, 18(4):1065–1081, 1997.